A Framework for Monte-Carlo Tree Search on CPU-FPGA Heterogeneous Platform via on-chip Dynamic Tree Management

Yuan Meng
ymeng643@usc.edu
University of Southern California
Los Angeles, CA, USA

Rajgopal Kannan
rajgopal.kannan.civ@mail.mil
DEVCOM US Army Research Lab
Los Angeles, CA, USA

Viktor Prasanna
prasanna@usc.edu
University of Southern California
Los Angeles, CA, USA

ABSTRACT

Monte Carlo Tree Search (MCTS) is a widely used search technique in Artificial Intelligence (AI) applications. MCTS manages a dynamically evolving decision tree (i.e., one whose depth and height evolve at run-time) to guide an AI agent toward an optimal policy. In-tree operations are memory-bound leading to a critical performance bottleneck for large-scale parallel MCTS on general-purpose processors. CPU-FPGA accelerators can alleviate the memory bottleneck of in-tree operations. However, a major challenge for existing FPGA accelerators is the lack of dynamic memory management due to which they cannot efficiently support dynamically evolving MCTS trees. In this work, we address this challenge by proposing an MCTS acceleration framework that (1) incorporates an algorithm-hardware co-optimized accelerator design that supports in-tree operations on dynamically evolving trees without expensive hardware reconfiguration; (2) adopts a hybrid parallel execution model to fully exploit the compute power in a CPU-FPGA heterogeneous system; (3) supports Python-based programming API for easy integration of the proposed accelerator with RL domain-specific bench-marking libraries at run-time. We show that by using our framework, we achieve up to 6.8× speedup and superior scalability of parallel workers than state-of-the-art parallel MCTS on multi-core systems.

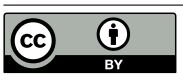
CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; • **Theory of computation** → Theory and algorithms for application domains; • **Computing methodologies** → *Game tree search.*

KEYWORDS

MCTS, AI Acceleration, Reinforcement Learning, Heterogeneous Computing

ACM Reference Format:
Yuan Meng, Rajgopal Kannan, and Viktor Prasanna. 2023. A Framework for Monte-Carlo Tree Search on CPU-FPGA Heterogeneous Platform via on-chip Dynamic Tree Management. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543622.3573177



FPGA '23, February 12–14, 2023, Monterey, CA, USA

ACM ISBN 978-1-4503-9417-8/23/02.
https://doi.org/10.1145/3543622.3573177

1 INTRODUCTION

Monte Carlo Tree Search (MCTS) utilizes a search tree along with random simulations to find optimal actions performed in an environment to maximize rewards. It is a general technique used in many domains, for example, Constraint Satisfaction Problems [18], Computer Games [19], and Neural-Architecture Search [22].

Tree-parallel MCTS methods have been proposed with the goal of enabling high-throughput simulations without negatively affecting the algorithm performance (i.e., domain-specific rewards that can be achieved with a fixed number of simulations) [3, 13, 19]. However, it is challenging to efficiently implement Tree-parallel MCTS on shared memory multi-core systems. This is because parallel workers synchronizing on a shared tree lead to frequent memory accesses, and MCTS in-tree operations have low arithmetic intensity. High-latency memory accesses cannot be hidden on shared-memory and data parallel architectures (e.g., CPU and GPU). As a result, in-tree operations become a bottleneck that hinder performance scalability to large number of workers on multi-core systems.

Because of the heterogeneity in the compute and memory characteristics of MCTS primitives such as simulations and in-tree operations, heterogeneous platforms consisting of both spatial architecture (i.e., FPGA) and general-purpose data-parallel architecture (i.e., multi-core CPU) should be exploited to efficiently accelerate MCTS. Existing works such as [14] use CPU-FPGA heterogeneous platform for parallel MCTS, where the FPGA acceleration of in-tree operations alleviates the memory bottleneck that otherwise occurs on CPUs to improve the system performance.

However, the state-of-the-art FPGA acceleration of MCTS requires a strictly static association between the topological ordering of tree nodes with the on-chip memory addresses. This severely limits the supported tree height and constrains the asymptotically growing characteristics of MCTS trees, thus affecting the domain-specific performance of MCTS algorithms. It is challenging to support runtime dynamic memory allocation on FPGA due to the nature of static memory assignation in FPGA bitstreams, while hardware reconfiguration lead to large time overhead.

In this work, we address the above challenges and propose the first CPU-FPGA MCTS acceleration framework that supports dynamic tree management without run-time FPGA reconfiguration. We achieve high performance by algorithm-hardware co-optimizations for the in-tree operations. Our framework maps the MCTS memory components and schedules MCTS primitives to concurrently exploit the compute power provided by both CPU and FPGA. The framework also provides an easy-to-use software API to reduce the effort for developing the application-agnostic accelerator and interfacing with high-level benchmarking libraries. Our original contributions are:

- To support arbitrary dynamic accesses by all the in-tree operations with minimal area overhead, we propose an accelerator design with an custom Butterfly-based Interconnection between the computing units and the memory banks;
- Based on our interconnection design, we propose an on-chip memory bank assignment algorithm for MCTS tree construction to minimize the runtime bank conflict during all the in-tree operations;
- To enable efficient hardware mapping and end-to-end execution, we develop a framework consisting of:
  - Accelerator Generator that decides the hardware configuration based on algorithm and benchmark parameters;
  - A hybrid parallel execution model that concurrently exploit the compute power of both CPU (data parallelism) and FPGA (pipeline parallelism);
  - Python-based API encapsulating the FPGA in-tree accelerator to make our design portable to state-of-the-art RL benchmarking libraries;
- Evaluation on widely-used game benchmarks show that our framework achieves up to 6.8× throughput improvement than the CPU implementations, and better algorithm performance than the state-of-the-art CPU-FPGA implementation of Tree-Parallel MCTS.

2 BACKGROUND

2.1 Monte-Carlo Tree Search

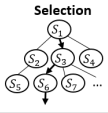
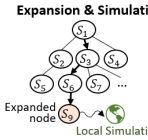
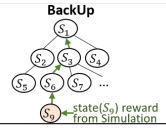
Monte Carlo Tree Search (MCTS) is a model-based Reinforcement Learning algorithm [3]. It is performed by an agent iteratively to plan the best sequence of actions with the goal of maximizing the cumulative reward. In the MTCS tree $\langle \mathcal{E}, \mathcal{S} \rangle$, a node $s \in \mathcal{S}$ records an environment state, an edge $e(s, \hat{s}) \in \mathcal{E}$ denotes an action taken at state s. At each agent step, the agent decides its action by constructing an asymmetric partial decision tree using one or multiple workers. Table 1 shows the operations performed by a worker in each iteration of the tree construction. In practice, in each agent step, the Selection, Expansion, Simulation, and BackUp phases are repeated until an iteration budget is reached [3, 14]. This budget is quantified as X that denotes the maximum number of recorded nodes in the tree (i.e., size of the MCTS tree).

2.2 Operations of Tree-Parallel MCTS

In parallel implementations of MCTS, multiple workers serve the same agent to speed up the agent decision process. Various approaches to parallelize MCTS have been proposed (Sec. 3.1). In this work, we target the popular Tree-Parallel MCTS method. The Tree-Parallel MCTS incorporates virtual-loss (VL) based in-tree operations. Specifically, after a worker selects an edge, a VL is subtracted from uct of selected edge to lower its weight and encourage other workers to take different paths. It also creates read-after-write dependencies between workers accessing the same tree node during Selection. VL is recovered later in BackUp. We define in-tree operations as all the arithmetic operations that access the tree in Selection (including applying VL), Expansion, and Backup:

- Selection (traversed edges E_t):
  - Read from tree levels: $s \leftarrow \underset{\hat{s} \in \text{Children}(s)}{\arg\max} \{uct(s, \hat{s})\}$
  - Apply Virtual Loss (VL): $uct(s, \hat{s}) -= VL$;
  - Node jumping: $s \leftarrow \hat{s}$; $E_t[i]$.append$(s, \hat{s})$;
- Node Insertion in Expansion (s'):
  - InsertNode(s'), InsertEdge(s, s');
- BackUp (E_t, reward R):
  - tree.UpdateEdges(R, VL, $uct(E_t)$)

Table 1: MCTS phases

Operations	Description
Selection 	The **Selection** phase iteratively selects $s \leftarrow \underset{\hat{s} \in \text{Children}(s)}{\arg\max} \{uct(s, \hat{s})\}$ until reaching a leaf, where $uct(s, \hat{s}) = V_{\hat{s}} + \beta\sqrt{\frac{\ln N_s}{N_{\hat{s}}}}$* [3].
Expansion & Simulation 	When leaf node s is selected, the **Expansion** phase runs a 1-step simulation from s to reach a new state, inserts a new child node s' to represent the new state, and creates an edge (s, s'). The **Simulation** phase runs local simulation from s' until termination, and returns a reward V.
BackUp 	The **BackUp** phase uses the received V to update the $V_{\hat{s}}$ term of uct for all the traversed edges during Selection.

Note: * $V_{\hat{s}}$ is the average expected reward that can be received through $\hat{s}$, and N_s ($N_{\hat{s}}$) denotes the number of times the nodes s ($\hat{s}$) has been visited. $uct(s, \hat{s})$ is the weight of the edge $(s, \hat{s})$. β is a parameter controlling the tradeoff between exploitation (first term) and exploration (second term) [3].

2.3 Performance Analysis and Challenges

2.3.1 Acceleration on general-purpose processors. The MCTS system throughput is described as the number of worker-Iterations performed Per Second, or IPS. A worker-Iteration is composed of the four phases (Table 1) conducted by a single worker. In tree-parallel MCTS, during each iteration, the number of in-tree operations and the number application-specific simulations are fixed. Therefore, IPS is upper-bounded by:

$$IPS_{\text{upper bound}} = min(PT_{sim}, PT_{in-tree}) \quad (1)$$

where PT_{sim} is the peak throughput of simulations (number of simulations performed per second) by all the workers, and $PT_{in-tree}$ is the peak throughput of the in-tree operations (number of Selection-Expansion-Backup performed per second) by all the workers. The simulation by all the workers are completely independent. Assuming there are p workers, PT_{sim} can be modeled as the total number of workers divided by the latency of single-worker simulation: $PT_{sim} = \frac{p}{T_{sim}}$. On the other hand, in the in-tree operations, all the workers are serialized at the root node for Selection and Update. We denote the amortized time interval between any two consecutive workers that access the root node as Itv. The peak throughput of the in-tree operations is thus bound by Itv: $PT_{in-tree} \leq \frac{p}{p \times Itv} = \frac{1}{Itv}$. This poses a constant upper-bound that prevent the system throughput from linearly increasing as the number of workers p scale up. In Fig. 1, we show the performance analysis based on Equation 1 using a classical control benchmark on a 128-core CPU. The line plots show the peak performance bound and the blue area is the range of

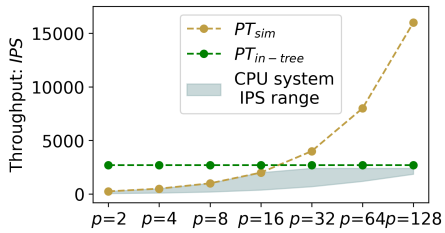


Figure 1: MCTS system performance on CPUs

actual *IPS* achieved. Note that the *IPS* for a specific p spans a range as it depends on the specific execution model (details discussed in Section 3.1).

2.3.2 FPGA acceleration of MCTS. A key perspective of efficiently balancing exploration-exploitation tradeoff in MCTS is the dynamic construction of its tree policy. The pattern of the tree growth is determined at runtime by the random simulations. While it is simple to perform dynamic tree management using runtime dynamic memory allocation on CPUs, this is a challenging task on FPGAs. This is due to the FPGA bitstream's nature of static memory assignation. A naive method of dynamically re-allocating memory blocks for the growing tree at runtime is through hardware reconfiguration, which causes unnecessary and large time overhead in the end-to-end MCTS execution. We are motivated to address this challenge by proposing the first dynamic MCTS accelerator design without the need for hardware reconfiguration.

3 RELATED WORK

3.1 Parallel MCTS: General-Purpose Processors

Several parallel MCTS algorithms have been developed to increase the throughput while reducing the negative impact on algorithm performance in terms of obtained rewards [4, 5, 11, 12]. Tree-Parallel MCTS and its variants benefit significantly from their superior algorithm performance compared with the other parallel methods [5, 13–15]. It has been adopted in various successful applications such as Go [20], Dou-di-zhu [23], and Atari games [13]. Therefore, Tree-Parallel MCTS is our target parallel approach for this work.

Existing Tree-Parallel MCTS on CPU can be categorized into two parallel execution models: multi-threaded tree traversal [5] and single-thread tree traversal [13].

- In multi-threaded tree traversal, each worker accessing the tree is assigned a separate thread, and local mutex at each tree node is used for accessing the shared tree. The main disadvantage of this method is that multiple threads communicate through the DDR memory which lead to high Itv dominated by DDR access time (hundreds of CPU cycles [1, 7]).
- In single-thread tree traversal, only a single master thread is assigned for performing in-tree operations exclusively, and multiple worker threads perform simulations exclusively. It has the advantage of low-latency memory access time since the tree can be managed in the local memory (e.g. last-level cache). It also achieves higher IPS than multi-threaded tree traversal, because the in-tree operations can be overlapped with simulations. However, Itv between workers is still large because all the workers are serialized, and the system performance cannot scale well even with a small number of workers (as shown in Figure 1, the master thread for in-tree operations becomes the bottleneck at $p = 16$).

In this work, we are motivated to achieve better system throughput and scalability compared with the existing implementations discussed above.

3.2 Hardware-Accelerated MCTS

[8, 17] design Blokus Duo Game solvers on FPGA that uses MCTS. Their accelerators target Blokus Duo game only and implement the simulator circuit on FPGA. It is difficult for their designs to generalize to various applications due to the lack of general-purpose simulators provided by CPU processors. [14] proposed to accelerate MCTS in CPU-FPGA heterogeneous systems, and developed FPGA accelerator for in-tree operations. However, the accelerator design in [14] requires static memory allocation for a full tree at compile time. This is because it assumes a static one-to-one association between the topological ordering of tree nodes with the on-chip memory addresses. As the memory requirement for the full tree increases exponentially wrt the tree height, the supported tree height is extremely limited on FPGAs which typically have limited on-chip resources. This constrain the asymptotically growing characteristic the tree, thus affecting the domain-specific algorithm performance of MCTS algorithms. In summary, none of the existing FPGA design support dynamic tree management which is critical in achieving high algorithm performance. In this work, we aim to bridge this gap by supporting dynamic tree management while maintaining high system throughput.

4 ACCELERATOR DESIGN

4.1 Overview

4.1.1 Data Structure and Operations. The MCTS tree is maintained on-chip of the FPGA accelerator. In the MCTS tree data structure, each node is associated with an ID based on insertion order, its number of visits, and the average reward gained by visiting it. Each edge has a parent ID, a child ID and a weight (UCT value). Assuming there are p workers, the accelerator performs all their in-tree operations (BackUp, Selection, and Node Insertion as listed in Section 2.2). Note that the application-specific environmental states are stored in the CPU memory rather than FPGA memory, and the rest of the Expansion phase including 1-Step simulation and environmental state management are also performed on the CPU instead of the FPGA (further discussed in Sec. 5.1).

4.1.2 Accelerator Overview. The overview of the accelerator is depicted in Fig. 2. The key idea of the accelerator design is to exploit pipeline parallelism among the workers that propagate through multiple stages, each stage operating on a certain tree level stored in on-chip SRAM. Assuming the maximum tree height is D, a pipeline is allocated with D pipeline stages, each stage equipped with an Inserter, a Selector and an Updater corresponding to operations on a tree level. Worker requests for the in-tree operations are streamed into the compute units (Inserter, Selector or Updater) from the PCIe Interface. Upon the completion of Selection and Node Insertion requests, The pipeline outputs requests for simulation back to the

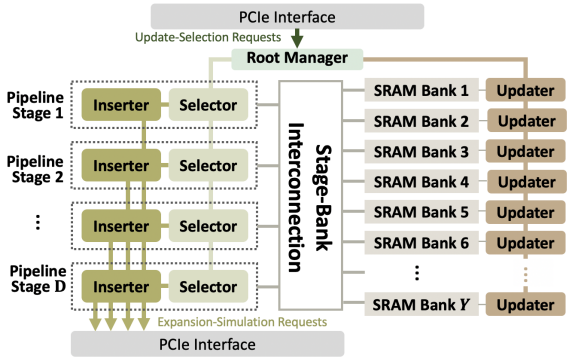


Figure 2: Accelerator Design: Overview

PCIe. The pipeline require read/write accesses to Y on-chip SRAM banks that store the tree through a custom Stage-Bank Interconnection. Each DRAM bank stores multiple tree node entries. The content of a node entry s stored in the SRAM banks is listed in the following, assuming F is the fanout of the tree:

- $s.ID$, number of visits $N(s)$
- $[uct(s, s'_0), ..., uct(s, s'_F)]$
- [Bank Index(s'_0), ..., Bank Index(s'_F)],
  [Node Index(s'_0), ..., Node Index(s'_F)]

In the above list, $N(s)$ and uct are updated by the Selectors; Bank Index (s) denotes the ID of the SRAM bank that stores node entry s, and Node Index(s) denotes the address of node entry s in its bank. Bank Indices and Node Indices are only assigned once by Inserters in each MCTS agent step.

As motivated in Sec. 2, the key objectives of the accelerator design are to (1) support dynamic run-time accesses to the tree nodes by the pipeline, and (2) ensure high performance by minimizing the Itv between workers, where Itv is composed of the propagation time in the Stage-Bank Interconnection and the Selector Compute time (Section 4.3). We realize objective (1) through a custom interconnection as discussed in Section 4.2. In Objective (2), we develop a novel bank assignment algorithm for tree construction to improve the throughput of in-tree operations by reducing the bank conflicts in the interconnection, and several hardware optimizations to improve the time performance of the Selector.

4.2 Stage-Bank Interconnection

For completely run-time dynamic tree management, the shape and topological ordering of tree nodes are not known at compile time, and are different for every MCTS agent step. For this reason, it is not practical to determine a fixed access pattern from the pipeline stages to the SRAM banks storing the tree nodes at compile time. Therefore, to avoid expensive run-time re-configuration between consecutive MCTS agent steps, an all-to-all interconnection between the stages and SRAM banks is required to support arbitrary run-time access patterns. An intuitive solution to meet this requirement is to build an all-to-all broadcast network that routes from all the stages to all the banks. However, this solution is not scalable to large or deep trees. Assuming the total number of SRAM banks used to store the tree is Y, the maximum height supported for the MCTS tree is D, the total area consumption of the all-to-all fully-connected network is $O(DY)$. This is impractical to implement on typical FPGA devices with limited on-chip resources. For example, such a design cannot be produced even for a tree height $D = 32$ when targeting a small MCTS tree stored in 256 SRAM banks.

We propose an area-efficient solution that can be better scaled to MCTS tree with large heights, while maintaining the capability of all-to-all interconnection for fully dynamic accesses by the compute units. The design of our Stage-Bank Interconnection follows a butterfly network pattern, as shown in Fig. 3.

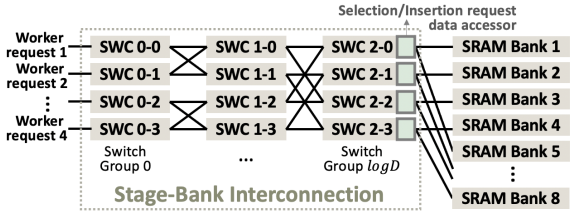


Figure 3: Example Custom Butterfly-based Interconnection: $D = 4, Y = 8$

Algorithm 1 Stage-Bank Routing from stage $i, i \in [0, D)$

Input: Binary representation of target SRAM bank index idx;
Output: A sequence of request-routing actions to SWC$\{x, y\}$ with the goal of accessing bank idx, where $x \in [0, \log D)$; $y \in [0, D)$;

```
SWC{x, y} ← SWC{0, i}
for x ∈ [0, ..., log D − 1) do                    ▷ bit 0 (log D − 1) is the LSB (MSB)
    if idx.bit[x]== 0 then                        ▷ Route Up
        if 0 <= y%(2^(x+1)) < 2^x then SWC{x, y} ← SWC{x + 1, y};
        else: SWC{x, y} ← SWC{x + 1, y − 2^x};
    else                                          ▷ Route Down
        if 2^x <= y%(2^(x+1)) < 2^(x+1) then SWC{x, y} ← SWC{x + 1, y};
        else: SWC{x, y} ← SWC{x + 1, y + 2^x};
```

The first part of the Stage-Bank Interconnection is a complete butterfly network with D input ports and D output ports that communicate the in-tree operation requests. The second part is a broadcast network from D switches to Y banks, which can be viewed as D independent 1−to−$\lceil \frac{Y}{D} \rceil$ connection units, and they facilitate processing the in-tree operation requests by providing read/write accesses to the memory banks. Our design reduced the area requirement to $O(D \log D + Y)$ while supporting the any-to-any fully connected access pattern which is essential for dynamic tree management. The routing algorithm is shown in Algorithm 1. Note that while our implementation of the Stage-Bank Interconnection can handle routing congestions, these congestions could negatively affect the throughput of in-tree operations. We further discuss optimizations to alleviate such performance degrade in Section 4.3.

4.3 Algorithm-Hardware Co-Optimizations

With the basic design discussed above, we derive the Itv between consecutive workers performing in-tree operations as the interval between consecutive workers making Selection requests. This is the sum of interconnection propagation time and the Selector compute time (i.e., latency for F−way comparison and applying virtual loss, where F is the fanout of the MCTS tree). Note that for both Node Insertion and Update requests, the interval between workers is much smaller than that for the Selection requests. This is because neither the Node Insertion nor the Update have RAW dependency between accessing different tree levels, such that $O(1)$ interval between workers can be easily achieved. On the other hand, the

Selection request of a subsequent worker can only be processed after the completion of $f-$way comparison and virtual loss update by its previous worker, so the overall Itv is equivalent to the interval between consecutive workers' Selection requests.

In the formulation of Itv, the interconnection propagation time can be further decomposed into single-worker latency and overhead from butterfly network congestions. We show the time complexity $T()$ and the worst-case time complexity $O()$ of these components of Itv in Equation 2:

$$\begin{aligned} Itv &= T_{\text{interconnection}} + \underbrace{T_{\text{selector}}}_{O(F)}, \\ \text{where } T_{\text{interconnection}} &= \underbrace{T_{\text{butterfly}}}_{T(\log D)} + \underbrace{T_{\text{congestions}}}_{O(D)} \end{aligned} \tag{2}$$

Based on the above analysis, we show our novel algorithm-hardware co-optimizations for reducing the time complexity of T_{selector} (latency of computing F-way comparison, discussed in Section 4.3.2) and $T_{\text{congestions}}$ (latency overhead from congestion of multiple Selection requests on the same interconnection switch, discussed in Section 4.3.1).

4.3.1 Dynamic Node Insertion Algorithm: Minimizing Interconnection Propagation Time. A potential bottleneck in the Itv time complexity is the time overhead from handling congestions by serializing the Selection requests at a certain bank or interconnection switch. The data layout of the tree (i.e., the bank assignment of each node entry during Node Insertion) plays a critical role in the number of congestions in the butterfly network during the process of Selection requests. In a basic scenario where the node entries are simply stacked into the banks one by one in their insertion order without any constraints, it is possible for node entries belonging to different tree levels to be stored in the same bank. In this case, all the D Selection requests in the Selection pipeline could collide on the same output switch in the butterfly network (although they are all independent requests by different workers) such that $T_{\text{congestions}} = D$ cycles. To avoid such scenario, we first put a constraint on the Node Insertion logic to ensure that nodes inserted on different tree levels cannot share the same bank ($Constraint1$). While this avoids the scenario of requests colliding on the same bank, it cannot avoid all the switch collisions, since multiple banks are connected to the same output port of the butterfly network part in the Stage-Bank Interconnection. To minimize $T_{\text{congestions}}$ for any given number of stages and SRAM banks, we propose a bank assignment algorithm for Node Insertion that minimizes the total number of switch congestions in Selection, based on the butterfly network properties, as shown in Algorithm 2. The intuition behind the proposed algorithm is as follows: We keep track of all the established routing paths using a scratchpad. When a new routing path is constructed by inserting a node into a new bank, we use the scratchpad information to select the bank that minimizes the total number of potential congestions with existing routing paths.

Our proposed bank assignment algorithm avoids exhaustively checking every pair of stage-bank connections. This is done by taking advantage of the recursive property of butterfly network structure and its routing algorithm 1: at every switch group (i.e. interconnection layer) x, the request from a given input port src can only collide with the request from another input port $src_{potential\_conflict}$ if bit x is the only different bit in the binary representations of both input ports (Algorithm 2 line 10), and the said collision can be checked by comparing bit $\log D - x$ of their destination output ports (this is same as bit $\log D - x$ of the target bank index, Algorithm 2 line 11-13). We denote the number of input/output ports to the Stage-Bank Interconnection as D (this is equivalent to the number of pipeline stages and the tree height), and the total number of nodes in the tree as X. Each node insertion into an existing bank takes 1 cycle (Algorithm 2 line 3). Each node insertion identifying a new bank assignment takes $O(Y \log D)$ cycles. Overall, for a complete tree construction in a MCTS agent step, the amortized time complexity of the bank assignment algorithm for each node is $\frac{(1+Y)Y \log D}{2} \times \frac{1}{X}$ cycles. The scratchpad memory $BankInfo$, $StageInfo$ are dynamically filled and used by the bank assignment logic. Their memory overhead are O(Y) for $BankInfo$ and O($DY/2$) for $StageInfo$. In typical MCTS benchmarks, D ranges from 8 to 32 and X ranges from 500 to 10K. Based on these parameters, the amortized single Node Insertion latency is only 1.02 to 20 cycles. Overall, our bank assignment algorithm optimization trade for low Itv during Selection by introducing latency overhead during Node Insertion. Algorithm 2 benefits the system throughput by reducing Itv between workers. Although it has the tradeoff of increasing the Node Insertion latency, this latency can be hidden in the heterogeneous system using our parallel execution model (Section 6.3).

Algorithm 2 Bank Assignment Algorithm for Node Insertion

Input: Node Insertion stage (tree level) id src;
Input: Scratchpad Memory $BankInfo$[bank id]={stage id, $bool\ full$}; ▷ $BankInfo$ tracks the tree level (stage) of the nodes stored in a bank
Input: Scratchpad Memory $StageInfo$[stage id]={list of destination port id}; ▷ $StageInfo$ tracks all the destination ports that a stage routes to
Output: The bank id $dest_{OPT}$ that minimizes $T_{\text{congestions}}$;

1: $current\_bank \leftarrow StageInfo[src][-1]$ ▷ Most recent accessed bank
2: **if** $!BankInfo[current\_bank].full$ **then**
3: $dest_{OPT} \leftarrow current\_bank$; Update $BankInfo[dest_{OPT}].full$
4: ▷ Inserting to existing bank under $Constraint1$
5: **else**
6: $min\_count \leftarrow \infty$
7: **for** $Bank_{Candidate}$ in $BankInfo$ with stage id==Null **do**
8: $congestion\_count \leftarrow 0$
9: **for** $x \in [0, ..., \log D - 1)$ **do** ▷ bit 0 is the LSB
10: $src_{potential\_conflict} \leftarrow src$.toggle$[bit(x)]$
11: **if** $\exists Bank_{conflict} \in StageInfo[src_{potential\_conflict}]$
12: such that $Bank_{conflict}.bit(\log D - x) ==$
13: $Bank_{Candidate}.bit(\log D - x)$ **then**
14: $congestion\_count++$
15: ▷ Identified congestion at Switch Group x
16: **if** $congestion\_count <= min\_count$ **then**
17: $min\_count \leftarrow congestion\_count$
18: $dest_{OPT} \leftarrow Bank_{Candidate}$
return $dest_{OPT}$

4.3.2 Hardware Optimization: Minimizing Selector Compute Time. Given a tree with Fanout F, each Selector can identify the best child node in F cycles using one comparator. To reduce T_{selector} and improve the selector performance scalability to large F, we use a hierarchical comparison-lookup design for low latency. Specifically, we define a comparison-lookup factor f, and recursively divide the $F\ uct$ values into f groups until each group is of size $<= f$. Within each group, we obtain the maximum of $f\ uct$ values in a single cycle using a comparison-lookup unit. The design is shown in Figure 4. Each comparison-lookup unit has $C_2^f = \frac{f!}{(f-2)!2!}$ comparators. Each

comparator outputs a 1-bit comparison result of a unique pair of *uct* values. The concatenation of these results (a C_2^f-bit number) is used to index a Look-Up table (size $2^{C_2^f}$) that outputs the best child index $\hat{s}$ (with the maximum *uct* value). This design allows latency of $\lceil \log_f F \rceil$-cycle response to any changes in F input *uct* values. This design is allocated in every Selector to concurrently process the Selection requests by different workers. f should be tuned for the optimal performance within the FPGA resource constraint.

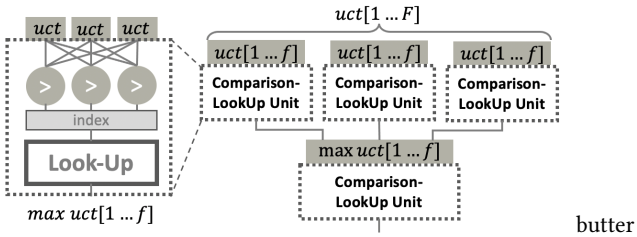


Figure 4: Example Comparison-LookUp Design: $F = 9, f = 3$

4.3.3 Other Optimizations.

Shift Register: After the Selection request accessing a certain tree level is completed at the output port of the Stage-Bank Interconnection, the workers propagate to the next pipeline stage by populating a shift-register array. The array collects the request processed at every tree level, and shifts by one tree level to serves as the input to the next-round Selection requests. Upon every shift-register operation of the array, one output Expansion-Simulation request is popped from the array and sent to the PCIe interface.

Memoization for BackUp: To eliminate the $O(D)$ overhead of sequentially back-tracing from leaf to node in BackUp of each worker, we allocate Y updaters, one associated with every bank, and let the update operation in all the tree levels perform in a data-parallel manner. A BackUp Memoization Buffer with size of $D - 1$ words is associated with each worker to memorize the node entries to be updated in BackUp during Selection. Thus for each worker, BackUp can be completed in 2 cycles.

5 FRAMEWORK SPECIFICATIONS

Our framework takes a benchmark software simulator, MCTS tree specifications, and CPU-FPGA platform specifications as inputs. It outputs the end-to-end mapping of the Tree-parallel MCTS on the given heterogeneous platform. It is composed of a parallel execution model that defines the primitives executed on each processor/accelerator and their interactions (Section 5.1), and a tool flow for generating FPGA bitstream and interfacing heterogeneous programming languages (Section 5.2).

5.1 Hybrid Parallel Execution Model

As introduced in Section 3.1, both existing CPU execution models (multi-threaded and single-thread tree traversals) have their tradeoffs. In this work, using our FPGA accelerator, we propose a hybrid parallel execution model that outperforms both existing execution models. The proposed hybrid execution model allows low-latency on-chip memory accesses and concurrent in-tree operations on FPGA that outperforms the multi-threaded CPU tree traversal, while keeping the advantage of high-throughput simulation of single-thread tree traversal. This is achieved by preventing the in-tree operations from occupying the simulation threads. Our execution model uses a task decomposition scheme consistent with existing work [14] to reduce the CPU-FPGA data traffic and FPGA on-chip memory consumption - the system dynamically maintains two memory components: The MCTS tree and State Table. The MCTS tree is stored on the FPGA (Section 4.1.2) and its node entries do not store the environment states. The State Table is stored in the CPU DRAM. It is implemented as a table with X entries (X is the tree size), where the index of each entry is a unique node index maintained in the MCTS tree, and the value is an application-specific environment state represented by that node.

The high-level heterogeneous system architecture of our framework is shown in Figure 6. We use a master-worker architecture to implement parallel MCTS under our hybrid execution model with the following considerations: First, the simulation operations during Expansion and Simulation phases are application-specific, worker-independent, and usually more time-consuming compared to the in-tree operations. So, they are implemented on the CPU in a data-parallel fashion using multiple worker threads. Second, a centralized Master FPGA Thread hosting a pipelined accelerator is dedicated for high-throughput in-tree operations using localized fast on-chip memory. This also prevents the in-tree operations from occupying and blocking the simulation processes.

5.1.1 Master FPGA thread and Simulation threads: Workflow. The execution workflow of the FPGA kernel, and each CPU thread is summarized in Figure 5. The Master FPGA Thread (1) serves as the host program for the FPGA accelerator, and (2) is used for coordinating and scheduling worker requests among the Simulation threads. The Master FPGA Thread is critical in overlapping in-tree operations with simulations. As shown in Figure 5-(b), the master process repeatedly executes the in-tree operations using the FPGA accelerator and assigns Expansion-Simulation tasks to different Worker Simulation Threads through the shared Exp-Sim request buffer (the buffer is implemented as a thread-safe queue). It collects the Update-Selection requests returned by the Worker Simulation Threads to update the MCTS tree statistics. The CPU-FPGA data communication and the communication between master and worker threads are asynchronous, allowing the in-tree operations and simulation by different workers to overlap. As shown in Figure 5-(a), after the pipelined processing of Update-Selection requests, the node IDs to be expanded can be generated before deciding the bank assignment of inserted nodes using Algorithm 2. Therefore, the Exp-Sim requests can be sent to CPU in the same time the Node Insertion is executed on the FPGA, hiding the additional overhead introduced by our algorithm optimization for the Node Insertion. The Worker Simulation Thread process is shown in Figure 5-(c). Note that The State Table allows fully data-parallel operations by all the simulation threads and does not incur additional synchronization between threads.

5.1.2 Dependency-Relaxed Task Scheduling. In synchronous tree-parallel MCTS, a barrier is put after the BackUp to make sure updates by all the workers are completed before the Selection in the next iteration can start. In practice, although this allows all the workers to access the most up-to-date statistics of the *uct* values

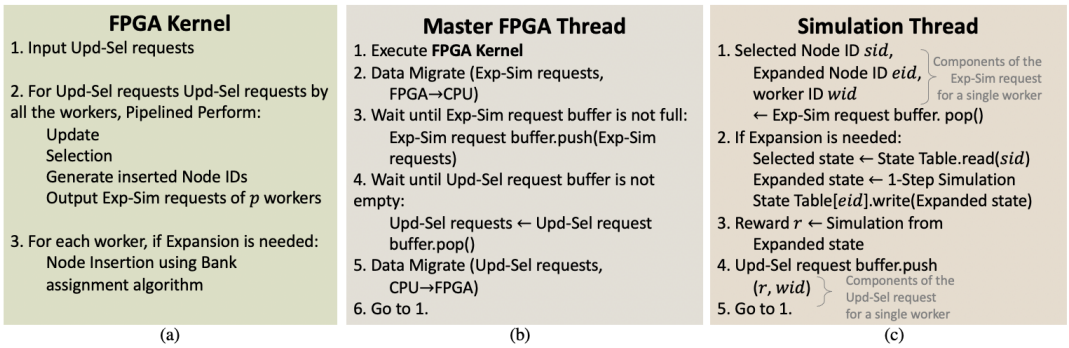


Figure 5: Hybrid Parallel Execution Model workflow. Exp-Sim denotes Expansionm and Simulation; Upd-Sel denotes Update (i.e., node updates in Back-Up) and Selection.

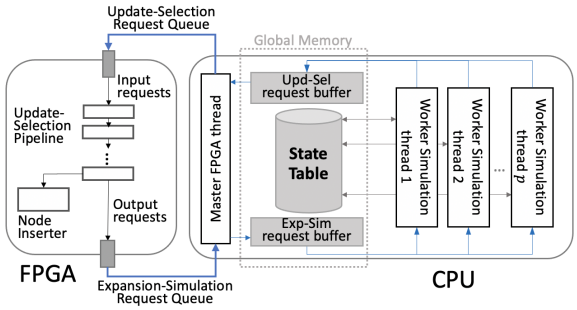


Figure 6: Heterogeneous System Overview. Global Memory is the CPU DRAM.

in the MCTS tree, it leads to idling of the FPGA hardware in the Selection pipeline. To alleviate this idling, we relax the dependency between different workers in adjacent iterations. Specifically, instead of having all the p workers waiting for the completion of BackUp by all the workers in the previous iteration, we only make sure each individual worker waits for the completion of BackUp by itself in the previous iteration before beginning Selection of its next iteration. This dependency-relaxed task scheduling can result in staleness of the tree policy by up to p updates behind, compared to the tree policy constructed with the barrier after BackUp. This effect on the algorithm performance is trivial because p (usually tens to hundreds) is usually much smaller than the total number of iterations in a MCTS agent step (up to tens of thousands).

5.2 Framework Workflow

Figure 7 summarizes the design tool flow in our framework. The tool is composed a DSE (Design Space Exploration) Engine, an accelerator code generator, and System API (Application Programming Interface) for interfacing between the FPGA kernel, the Master FPGA thread and the simulation threads in high-level language (Python) under our proposed hybrid execution model. The inputs to the framework include configuration files describing the MCTS Tree and the FPGA hardware given at compile time, and command line arguments specifying the benchmark environment and number of workers given at run time. The output is an executable Python program that performs the specified MCTS application on a CPU-FPGA platform. The DSE engine and the API are discussed in detail in Section 5.2.1 and 5.2.2, respectively.

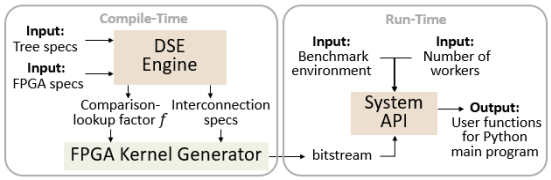


Figure 7: Framework Design Tool Flow

5.2.1 FPGA Accelerator Generation. The input MCTS tree specifications include the tree height limit D, fanout of the tree (i.e., action space) F, and the iteration budget (tree size limit) X. Our proposed FPGA accelerator has two major design parameters: the Stage-Bank interconnection configuration, M, and the Comparison-lookup factor, f. The interconnection configuration has two modes: "butterfly" specifies the custom Butterfly-based Interconnection (Section 4.2). "all-to-all" specifies an all-to-all connection between the stages and banks yielding the optimal (1-cycle) bank access latency. f is an integer factor (Section 4.3.2). The DSE engine determines these design parameters with the goal of minimizing Itv (Equation 2).

$$
\begin{aligned}
T_{\text{selector}} &= \log_f F \\
T_{\text{interconnection}} &= \begin{cases} \log_2 D & \text{if } M\text{="butterfly"} \\ 1 & \text{if } M\text{="all-to-all"} \end{cases} \\
R_{\text{selector}} &= \left(C_2^f \times R_{\text{comparator}} + R_{\text{Lookup}}\right) \sum_{n=0}^{\log_f F - 1} f^n \\
R_{\text{interconnection}} &= \begin{cases} (D \times \log_2 D + Y) \times R_{\text{buffer}} & \text{if } M\text{="butterfly"} \\ D \times Y \times R_{\text{buffer}} & \text{if } M\text{="all-to-all"} \end{cases}
\end{aligned} \tag{3}
$$

Based on the above models, our DSE engine search for the global optimal design point on a target FPGA that satisfies Equation 4:

$$
\begin{aligned}
&M, f = \text{argmin}\{D \times T_{\text{selector}} + T_{\text{interconnection}}\} \\
&\text{such that } D \times R_{\text{selector}} + R_{\text{interconnection}} < R_{\text{FPGA}} \\
&\text{where M} \in \{\text{"butterfly"}, \text{"all-to-all"}\}, f \in [1, F]
\end{aligned} \tag{4}
$$

In Equations 3 and 4, R_{FPGA} denotes the available set of FPGA resource (DSP, LUT, SRAM, etc.) and R_{module} denotes the resource consumption of a module. T_{module} denotes the latency (number of FPGA cycles) to process a request using the module. After the Stage-Bank interconnection configuration and the Comparison-lookup factor are determined, they are used by an FPGA Kernel Generator script to produce the HLS code and compile the code into an FPGA bitstream.

5.2.2 System API. Our framework need to link across heterogeneous programming languages of CPU and FPGA. Specifically, state-of-the-art AI bench-marking simulators executed on CPU need to be invoked in high-level libraries using Python [2, 21], while our FPGA Kernel are described and hosted using HLS (C++) code. Therefore, an API is needed for the CPU-FPGA runtime system to port the FPGA Kernel initiation and communication protocols into Python functions. Table 2 summarizes the API functions provided by our framework. The listed functions are called from a main program executed on the CPU master thread. For the functions interfacing with FPGA (rows 1, 3, 6, 7 of Table 2), we use the Pybind library [9] to develop Python wrappers for our HLS C++ host code.

Table 2: API functions

API Functions	Description
Init()	Initialize the platform with FPGA bitstream, Initialize the benchmarking environment for simulation
MCTS_Parameters()	Set the MCTS tree parameters and number of workers
LoadKrnlParameters()	Generate and Load the static content of the Comparison-LookUp Tables on FPGA
AssignExpSimTasks()	Check the parallel worker pool and execute thread-safe protocol for sending Expansion-Simulation requests from the master thread to a worker simulation thread
CollectSimTasks()	Check the parallel worker pool and execute thread-safe protocol for receiving Update-Selection requests from a worker simulation thread to the master thread
SendInTreeRequests()	Send the Update-Selection requests from the master thread to FPGA
ReceiveSimRequests()	Receive the Expansion-Simulation requests from FPGA to the master thread

6 EVALUATION

The objectives of our work are to support dynamic in-tree operations with low interval (Itv) between workers, and to improve the scalability compared to CPU-only systems. In the following subsection, we evaluate a). How the proposed dynamic tree management affects the performance of In-tree operations (Section 6.2); b) How the proposed parallel execution model affects the system throughput (Section 6.3); c). The MCTS algorithm performance using our framework (Section 6.4).

6.1 Implementation Specifications

Benchmark environments: We evaluate our framework on Atari games, a classic benchmark for evaluating reinforcement learning and planning algorithms [6]. We choose three benchmarks: Carnival, Pong, both with action space (i.e., fanout F of the tree) 6, and Alien with action space 18. For these games, both the Simulation and the 1-step simulation in Expansion use OpenAI-gym library. We set the MCTS tree size limit (X) as $10K$ for all our experiments as consistent with state-of-the-art implementations.

Platforms: Our CPU baseline experiments are conducted on an AMD EPYC 7763 64-Core Processor server with 2 sockets (256 hardware threads in total) at 1.5 GHz. The CPU-FPGA platform consists of the same CPU and a Xilinx Alveo U200 board [24] connected by PCIe. In all the experiments, p denotes the number of workers, each worker uses a CPU worker Simulation thread. We use two CPU-only baseline implementations that follow multi-threaded and single-thread tree traversal execution models, respectively. Both are implemented using the Python Multiprocessing class.

FPGA Implementation specifications: We develop the FPGA kernel template using High-Level Synthesis (HLS). We follow VITIS development flow [10] for bitstream generation. OpenCL [16] is used to implement the data transfer between the CPU and FPGA. The FPGA kernel code generator takes less than 3 seconds to generate the HLS code for any of our test cases.

The resource utilization of our accelerator for both benchmarks are shown in Table 3. Note that the Carnival and Pong benchmarks use the same hardware configuration because they have the same tree fanout F and tree size X. The resource consumption bottleneck is in LUTs since both the interconnection and the selector comparison-lookup modules compete for the LUT resources. Specifically, the LUT consumption in the butterfly interconnection $\propto D\log(D)$, and LUT consumption in the selectors also grows asymptotically with increasing F and D (Equation 3). The largest design that we can place on the target FPGA is the Alien benchmark with $F = 18, D = 32$, which uses up to $\sim$ 50% of the available LUTs.

Our design achieves 250 MHz operating frequency. For $D \leq 4$, all-to-all interconnection is chosen by the DSE engine. As D increases, butterfly-based interconnection is chosen. For both interconnection configurations, the critical path is in the switches of the stage-bank interconnection. The critical wire length is expected to grow with increasing D. For the largest D we can place on the target FPGA board ($D = 32$), the design is able to achieve single-cycle interconnection layer propagation operating at 250MHz. For larger D, inserting a register on the critical wire can help maintain high operating frequency.

Table 3: FPGA Resource Consumption

Benchmarks	SRAM	DSP	FF	LUT
Carnival, Pong	1.7∼1.88 MB (4.9\%∼5.4%)	289∼385 (4.9\%∼6.5%)	151∼204K (8.5\%∼11.5%)	121∼179K (13.9\%∼20.6%)
Alien	5.01∼5.43 MB (14.4\%∼15.8%)	273∼385 (4.7\%∼6.5%)	181∼401K (10.1\%∼22.5%)	162∼418K (19.2\%∼48%)

Note: $D = 8 \sim 32$ for all the benchmarks. Number of SRAM banks (Y) is set to 128 for all the test cases.

6.2 Performance of In-tree Operations

As discussed in Section 2.3 and 4.3, the serial time interval (Itv) between two workers sharing access to the tree determines the upper-bound of the system throughput when scaling to large number of workers. The lower the Itv is, the higher scalability is achieved.

Effect of Dynamic Algorithm Optimization: In Table 4, comparing the rows labeled "Ours (Dynamic)" with the rows "Ours (without Alg. 2)", it can be observed that our algorithm optimization reduces Itv to $\frac{1}{5}$ of its baseline value. The tradeoff of this optimization is in the longer time overhead for node insertion; however, this can be hidden in our parallel execution model (see Section 6.3).

Comparison with state-of-the-art: We compare Itv of our design with existing work [14]. [14] developed a pipelined accelerator for in-tree operations with static memory allocation for a full tree. We point out that the key difference of this work compared with [14] is that our proposed design is capable of supporting dynamic construction of arbitrary-shaped tree at run-time. We test our accelerator and the baseline accelerator ([14]) by feeding synthetic sequence of in-tree operations generated in an entire episode of agent steps, with various tree shape constraints parameterized by the tree height limit D. As shown in Table 4, our design supports various shapes since we do not set compile-time constraints on the

one-to-one correspondence between the topological location of the tree nodes and SRAM addresses.

Higher Itv on Alien compared with the other benchmarks is due to its larger tree fanout, making the comparison-lookup latency higher. Our design shows higher Itv on narrower trees (large D) compared with wider trees (small D). This is because the butterfly-based interconnection leads to $Itv \geq \log D$ cycles. On the other hand, because [14] only pre-allocates SRAM banks for a complete tree and constrains the nodes into their corresponding SRAM addresses, the maximum tree height is limited to $\log_F S$, where S is the largest number of node entries that can be stored on-chip. This limit is as low as 8 for Carnival and Pong, and 4 for Alien. In summary, our design can support a larger variety of tree shapes with different tree depth limits, whereas [14] can only support small tree depth, as summarized in Table 4.

Table 4: Itv of Dynamic vs Static Tree Management

Benchmarks	In-Tree Ops Accelerator	Tree Shape Constraints		
		D=8	D=16	D=32
Carnival	Ours (Dynamic)	Itv=3.2	Itv=3.5	Itv=9.7
	Ours (w/o Alg. 2)	Itv=8.1	Itv=17.6	Itv=36.4
	[14] (Static)	Itv=2	No Support	No Support
		D=8	D=16	D=32
Pong	Ours (Dynamic)	Itv=3.3	Itv=3.7	Itv=9.4
	Ours (w/o Alg. 2)	Itv=8.5	Itv=16.9	Itv=34.1
	[14] (Static)	Itv=2	No Support	No Support
		D=4	D=16	D=32
Alien	Ours (Dynamic)	Itv=4.3	Itv=4.7	Itv=10.5
	Ours (w/o Alg. 2)	Itv=7.3	Itv=18.2	Itv=37.8
	[14] (Static)	Itv=3	No Support	No Support

Note: Itv is the average number of cycles over all the iterations in an agent step. The rows labeled "w/o Alg. 2" describes a baseline design without the bank assignment algorithm for node insertion (instead, a simple heuristic for inserting the node to the next empty bank is used).

Comparison with CPU baselines: Our framework using FPGA lead to additional communication overhead through PCIe, compared with CPU-only baselines. Therefore, we measure the end-to-end latency of in-tree operations including the PCIe data transfer time. The PCIe data transfer time is obtained using Xilinx Runtime (XRT) Profiler [25]. The CPU-only baselines include the multi-threaded tree traversal and single-thread tree traversal. We first observe that on the CPU, the multi-threaded implementation does not significantly outperform single-thread implementation. This is because threads must be serialized at root-level, where the root-child nodes must be protected by a mutex to ensure only one thread can access it at a time. The sequential time interval between pair of consecutive threads accessing the shared root-children is dominated by the high latency access to the CPU shared memory (DDR), which cannot be reduced by increasing p. On the other hand, the single-thread implementation only allows a master thread to access the tree, so the tree can be accessed with lower latency in its local memory (cache). This is at the cost of serializing the in-tree operations by all the workers. For both benchmarks, the FPGA accelerator leads to lower latency than CPU, and consistently shows higher speedup compared with CPU at larger number of workers. The PCIe overhead increases very little as p increases, because the reduced PCIe data transfer time is negligible compared with the fixed PCIe initiation latency (~ 0.04 ms).

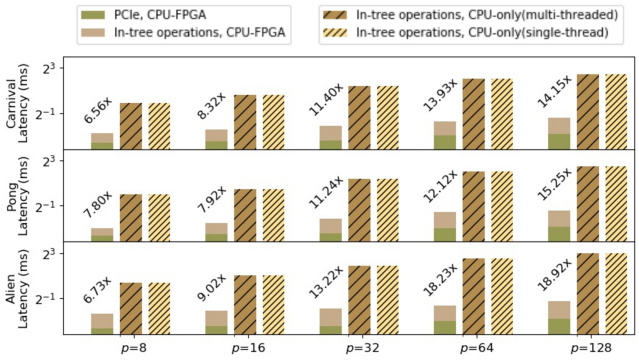


Figure 8: Latency of in-tree operations. Y-axes are in log scale for better visualization.

6.3 MCTS System Throughput

Figure 9 shows the timeline of the operations on the FPGA, the CPU master thread and one CPU worker Simulation thread in each iteration of our framework. We observe that the node insertion process with our bank assignment algorithm optimization (Algorithm 2) can be completely hidden by the simulation process. We also observe that the overhead for managing the request buffers and queues for communication between the master thread and worker simulation threads on CPU are small, and they can be overlapped with the simulation processes as they are fetched on-the-fly. This means that these overheads will not become bottlenecks that can hinder system scalability to large number of workers.

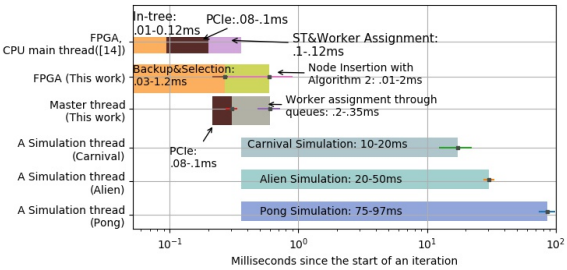


Figure 9: Timeline of the parallel execution in [14] and our framework. p=128.

The achieved system throughput in *IPS* (worker-Iterations processed per second) is plotted in Figure 10.

Comparison with state-of-the-art: We compare our system execution timeline with the state-of-the-art [14], as shown in Figure 9. Note that [14] adopts a different execution model where the PCIe communication and State Table accesses are blocking (implicit barriers are present before and after PCIe data transfer). On the other hand, our execution model allows overlapping the communication with the Selection and Node Insertion processes on FPGA. While the in-tree operation latency of our design is higher than those in [14], its effect on the system throughput is very small, since the accelerated FPGA kernels lead to small ($\leq$ 10%) overheads in each iteration. As a result, the achieved throughputs in [14] and in this work are close to the peak simulation throughputs $\frac{p}{T_{sim}}$.

Comparison with CPU baselines: In both CPU-only system execution models, the throughput can linearly scale up with increasing p until the total latency of serialized in-tree operations

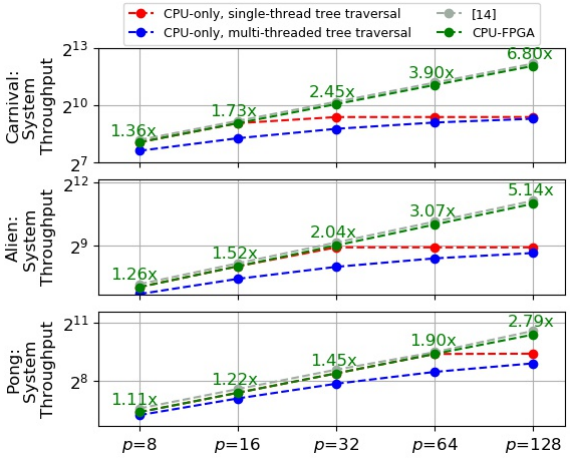


Figure 10: System Throughput Comparison. D=32.

become the bottleneck (For single-thread tree traversal, in-tree operations by workers are completely serialized. For multi-threaded tree traversal, in-tree operations by workers are serialized by the overhead of communicating root-level information across different threads through DDR memory). This threshold is p =16, 32 and 64 on the Carnival, Alien, and Pong benchmarks, respectively. The value of the threshold is affected by the ratio of the in-tree operations time to the simulation time. On the CPU, for benchmarks with lower simulation latency, the faster in-tree operations become the bottleneck as p scales up. For p larger than this threshold, the *IPS* no longer scales up. By reducing Itv between workers, the proposed hybrid parallel execution model that leverages FPGA acceleration alleviates the bottleneck imposed by in-tree operations. In our CPU-FPGA system, higher system throughput improvements are consistently observed for larger p, as evident in Figure 10. Overall, we obtain up to 2.8×, 5.14× and 6.8× higher throughput for the three benchmarks compared with the CPU-only baselines.

6.4 MCTS algorithm performance

The bar plots in Figure 11 show the total accumulative rewards gained using our framework, an existing CPU-FPGA baseline [14], and the CPU baseline with the same number of simulations. The tree height limit is set to $D = 32$ for both benchmarks in this work and the CPU baseline, while $D = 4(8)$ for Alien(Carnival) in the existing work [14].

Comparison with CPU baselines: For the CPU-only baseline, we only show the rewards from the single-thread tree traversal as it is very close to the rewards using multi-threaded tree traversal (the average difference is within 2%). Overall, our framework achieves similar algorithm performance in terms of rewards gained in an episode compared to the CPU baseline, with better scalability to large number of workers (shown by lower time per agent step in the line plots). We also show that the dependency-relaxation (Section 5.1.2) improves speed without significantly affecting algorithm performance. This is shown in Figure 11 by comparing our execution model (async.) with a synchronous version (sync.) that enforces the dependency of Selection upon all the worker Backups in the previous iteration.

Comparison with state-of-the-art: As shown in Figure 11, in the Alien and Carnival benchmarks, the rewards achieved by our framework are significantly higher than the rewards achieved in [14]. This is due to the ability of the proposed design to dynamically adjust the tree shape constructed. For the Pong benchmark, the rewards obtained over all the baselines do not show a significant difference; They are saturated at 21. This is because the agent wins the game and terminates it once it hits the score of 21 without letting the enemy hit the same score. We still notice a disadvantage of [14] in terms of the achieved score compared with this work and the CPU baselines. In our FPGA-based design, achieving higher algorithm performance comes at the cost of the additional overheads from interconnection routing and the node insertion algorithm. However, because the node insertion overheads can be completely hidden using our proposed execution model, the time per agent step achieved by our framework is very close to that of [14].

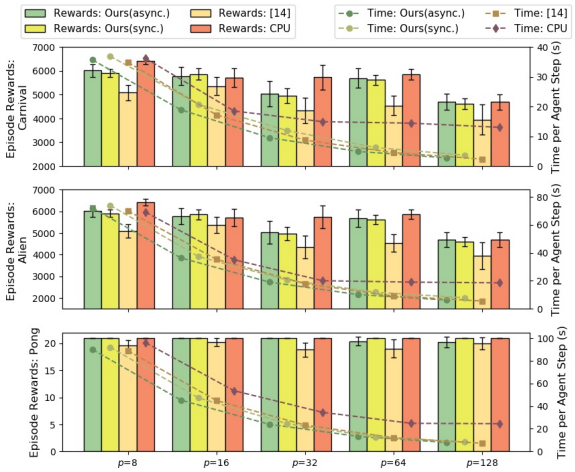


Figure 11: Rewards under various frameworks. async.(sync.) stands for execution with(without) dependency-relaxation.

7 CONCLUSION AND FUTURE WORK

In this work, we proposed an algorithm-hardware co-optimized accelerator design for in-tree operations in MCTS. Our design addressed the limitation in the state-of-the-art accelerator by supporting dynamic tree construction and management, while maintaining the high throughput and scalability with a hybrid parallel system execution model targeting CPU-FPGA heterogeneous platform.

Parallel MCTS also leads to many further research opportunities. For example, heterogeneous hardware acceleration can be exploited for deep learning guided MCTS where the system integrates DNN inference and training in the loop. Additionally, new MCTS parallelization approaches can be explored to reduce data race between workers in Tree-Parallel MCTS and enable efficient implementations on data-parallel and distributed platforms.

8 ACKNOWLEDGEMENT

This work is supported by the National Science Foundation (NSF) under grant numbers CNS-2009057, SPX-1919289, and in part by DEVCOM Army Research Lab (ARL) under ARL-USC collaborative grant DIRA-ECI:DEC21-CI-037.

REFERENCES

[1] AMD. 2019. *Zen Specification.* https://www.7-cpu.com/cpu/Zen2.html

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[3] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[4] Tristan Cazenave and Nicolas Jouandeau. 2007. On the parallelization of UCT. In *Computer games workshop.*

[5] Guillaume MJ-B Chaslot, Mark HM Winands, and HJVD Herik. 2008. Parallel monte-carlo tree search. In *International Conference on Computers and Games.* Springer, 60–71.

[6] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. 2014. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. *Advances in neural information processing systems* 27 (2014).

[7] Intel. 2018. *SkyLake Specification.* https://www.7-cpu.com/cpu/Skylake.html

[8] Ali Jahanshahi, Mohammad Kazem Taram, and Nariman Eskandari. 2013. Blokus Duo game on FPGA. In *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013).* IEEE, 149–152.

[9] Wenzel Jakob. 2022. *PyBind11.* https://github.com/pybind/pybind11

[10] Vinod Kathail. 2020. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 173–174.

[11] Hideki Kato and Ikuo Takeuchi. 2010. Parallel monte-carlo tree search with simulation servers. In *2010 International Conference on Technologies and Applications of Artificial Intelligence.* IEEE, 491–498.

[12] Hideki Kato and Ikuo Takeuchi. 2010. Parallel monte-carlo tree search with simulation servers. In *2010 International Conference on Technologies and Applications of Artificial Intelligence.* IEEE, 491–498.

[13] Anji Liu, Jianshu Chen, Mingze Yu, Yu Zhai, Xuewen Zhou, and Ji Liu. 2020. Watch the Unobserved: A Simple Approach to Parallelizing Monte Carlo Tree Search. In *International Conference on Learning Representations.* https://openreview.net/forum?id=BJlQtJSKDB

[14] Yuan Meng, Rajgopal Kannan, and Viktor Prasanna. 2022. Accelerating Monte-Carlo Tree Search on CPU-FPGA Heterogeneous Platform. In *2022 International Conference on Field-Programmable Logic and Applications (FPL).* IEEE.

[15] S Ali Mirsoleimani, H Jaap van den Herik, Aske Plaat, and Jos Vermaseren. 2018. Pipeline Pattern for Parallel MCTS.. In *ICAART (2).* 614–621.

[16] Aaftab Munshi. 2009. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS).* IEEE, 1–314.

[17] Ehsan Qasemi, Amir Samadi, Mohammad H Shadmehr, Bardia Azizian, Sajjad Mozaffari, Amir Shirian, and Bijan Alizadeh. 2014. Highly scalable, shared-memory, Monte-Carlo tree search based Blokus Duo Solver on FPGA. In *2014 International Conference on Field-Programmable Technology (FPT).* IEEE, 370–373.

[18] Baba Satomi, Yongjoon Joe, Atsushi Iwasaki, and Makoto Yokoo. 2011. Real-time solving of quantified csps based on monte-carlo game tree search. In *Twenty-Second International Joint Conference on Artificial Intelligence.*

[19] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

[20] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[21] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. 2018. Deepmind control suite. *arXiv preprint arXiv:1801.00690* (2018).

[22] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. 2020. Neural architecture search using deep neural networks and monte carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 9983–9991.

[23] Daniel Whitehouse, Edward J Powley, and Peter I Cowling. 2011. Determinization and information set Monte Carlo tree search for the card game Dou Di Zhu. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11).* IEEE, 87–94.

[24] Xilinx. 2020. *Alveo U250 Data Center Accelerator Card.* https://www.xilinx.com/products/boards-and-kits/alveo/u250.html

[25] AMD Xilinx. 2022. *XRT Profiling.* https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Profiling-the-Application