Parallelizing Covolutional Networks for Faster AlphaZero Inference

Kaustabh Paul Hanlu Xu
Department of Electrical and Computer Engineering, Carnegie Mellon University
Emails: kaustabp@andrew.cmu.edu, hanlux@andrew.cmu.edu
GitHub: `RedTorus/parallel_CNN_alpha_zero`

***Abstract*—Recent advances in deep reinforcement learning—exemplified by the AlphaZero paradigm—have shown that powerful convolutional neural networks (CNNs) can master complex board games through self-play. However, real-time human–AI interaction demands very low inference latency, which off-the-shelf GPU libraries do not always deliver when deployed in resource-constrained or highly parallel settings. In this project, we selected checkers as our target domain and set out to investigate how custom GPU kernels might accelerate the CNN-based inference stage. Rather than modifying the existing AlphaZero codebase directly, we designed and implemented a standalone inference pipeline in which every convolutional layer is executed via hand-tuned CUDA kernels. We then benchmarked our implementation against widely used GPU convolution routines under a variety of board-state inputs. Our results demonstrate consistent inference speedups of up to 4.9× over the baseline library calls fore some layers. These findings highlight the potential of specialized CUDA optimizations to improve the responsiveness of neural-network-driven game engines, and lay the groundwork for integrating similar techniques into full AlphaZero-style frameworks in future work.**

***Index Terms*—reinforcement learning, CUDA, inference acceleration, AlphaZero, checkers**

I. Introduction

Mastery of sophisticated games using self-play is now a benchmark for artificial intelligence, demonstrating the effectiveness of deep reinforcement learning and neural-guided search. Systems such as AlphaZero have shown that with the appropriate intersection of network architecture and planning, it is possible to achieve superhuman performance with no knowledge incorporated from humans [1]. While such feats have typically focused on learning efficiency as well as end-playing strength, there was less consideration of the practical requirements of executing such a system within real-time contexts.
The biggest real-world deployment limitation is currently the speed of neural network inference, i.e., convolutional layer evaluation at each decision boundary. For interactive applications where response speed is of utmost importance, the cost of conventional GPU inference can be a performance limiter. To fill the gap, specialized GPU programming approaches are investigated to speed up the inference part of AlphaZero-like pipelines.

Before diving into our approach and findings, we first review the key components of AlphaZero's design, focusing on how deep convolutional networks and Monte Carlo Tree Search work together during training and play.

Identify applicable funding agency here. If none, delete this.

II. Background

A. AlphaZero Workflow

AlphaZero combines a deep neural network with advanced look-ahead search to achieve superhuman performance. Its network is made of residual convolutional layers with two outputs: one predicts the probability of legal moves, and the other estimates the likely game outcome. When choosing a move, AlphaZero runs Monte Carlo Tree Search (MCTS) from the current position, balancing network predictions with exploration of uncertain moves. After hundreds or thousands of simulations, it builds a refined, high-confidence move choice far stronger than the raw network suggestion.

Learning happens in two phases: self-play and training. In self-play, AlphaZero plays against itself, recording positions, MCTS-based move probabilities, and final results. During training, the network updates to better match the improved move probabilities and more accurately predict outcomes. Over many cycles, the network and search process strengthen each other, allowing AlphaZero to master complex games without any human data or hand-crafted rules.

B. Checkers

For this project, we selected one specific board game, Checkers, to evaluate the architecture of the convolutional layer and design the parallelism strategy. The architecture of the convolutional layer is determined with considerable computational workloads to ensure efficient utilization of GPU capabilities.

A standard Checkers board consists of an 8×8 grid, where only the dark squares are used for moves. Each square is either empty or occupied by a piece. There are two players: Player 0 typically controls the white pieces, and Player 1 controls the black pieces. At any given point in a checkers game, each square has one of the five potential states, defined and implemented in the code as follows:

- kWhite: regular white piece ('o').
- kBlack: regular black piece ('+').
- kWhiteKing: kinged white piece ('8').
- kBlackKing: kinged black piece ('*').
- kEmpty: empty square (' ').

Consequently, a state of the checkers game can be represented by an input tensor of shape [5, 8, 8], corresponding to the 8×8 board dimensions and the five possible piece types, without batching.

C. Convolutional Layer Architecture

The architecture of the AlphaZero residual convolutional layer used for the checkers game is illustrated in Fig 1. It consists of three sequential convolutional block layers: the input block (ResInputBlock), the torso block (ResTorsoBlock), and the output block (ResOutputBlock).

The ResInputBlock receives a 2-D tensor of shape [1, 320] as its input, reshapes it into a 4-D tensor of shape [1, 5, 8, 8], and processes it with a convolutional neural network (CNN). This CNN uses 5 input channels, 128 output channels and 3 x 3 filters. The output of the convolution is then normalized via batch normalization and activated using the Rectified Linear Unit (ReLU) function. The resulting tensor is passed to the ResTorsoBlock.

The ResTorsoBlock forms the core of AlphaZero's convolutional processing. It first clones the input tensor to serve as the residual block. The tensor is then passed through a CNN with 128 input channels, 128 output channels, and 3 x 3 filters, followed by batch normalization and ReLU. It proceeds through a second CNN of the same configuration and another batch normalization operation. The output is then added with the original residual block, and the sum is passed through another ReLU. AlphaZero defines a parameter, nn_depth, to specify the number of such ResTorsoBlocks. In our setup, nn_depth is set to 5, so the tensor undergoes this residual block operation five times.

Finally, the output tensor is passed to the ResOutputBlock. The ResOutputBlock constructed two tensors in two separate branches, one for predicting the value and the other for predicting the policy logits. For the value head, the tensor first passes through a CNN with a 1 x 1 filter that reduces the number of channels from 128 to 1. The output is normalized using batch normalization and activated with ReLU, and then reshaped into a 2-D tensor of shape [1, 64]. The 2-D tensor is processed by two sequential linear transformations with a ReLU operation in between. The final output passes through a hyperbolic tangent (Tanh) activation function, resulting in a 2-D tensor of shape [1, 1] representing the value output. For the policy head, a similar initial processing is applied: a CNN with a 1 x 1 filter reduces the channels from 128 to 2, followed by batch normalization and ReLU. After reshaping into a 2-D tensor of shape [1, 128], the tensor undergoes a linear transformation and then performed by a `torch::where` operation at the end to mask invalid moves.

We will discuss our parallelism strategy in detail based on this specific convolutional layer architecture in the next section.

III. Parallelism Design

A. Parallelize Input Residual Layer

The Input-Residual layer processes a single input feature map of dimensions $[1, C_{in} = 5, H = 8, W = 8]$ to produce an output of dimensions $[1, C_{out} = 128, H = 8, W = 8]$ via 128 independent 3×3 convolutions (stride = 1, padding = 1). To eliminate global-memory accesses and prevent shared-memory bank conflicts on a 32-bank Turing SM, the $5\times8\times8 = 320$ input activations are staged into a two-dimensional shared-memory buffer: $input[10][33]$

The 320 values are logically arranged as 20 rows of 32 columns, with a +1 "pad" in the column stride (33) so that the physical row stride (33) is coprime with the number of banks (32). An access at coordinates (r,c) maps to bank $(r33+c)mod32$ and since $gcd(33, 32) = 1$, no two threads within the same warp contend for the same bank.

The weight tensor consists of 128 distinct kernels of shape $[5, 3, 3]$. Exactly 128 thread-blocks—one per output channel—are launched, so that each block requires only its own 45 weight values. These are loaded into a one-dimensional shared-memory array: $W_{kernel}[45]$
Since 45 is not a multiple of 32, sequential addresses naturally distribute across banks without conflict; accordingly, no padding is necessary.

Once both input and weight tensor subsection reside in shared memory, each of the block's 64 threads computes exactly one output pixel by performing the 5×3×3=45 multiply–add operations. All shared-memory reads complete in a single cycle, allowing the 45 FLOPs of computation to fully overlap with data delivery and thereby maximize throughput for the Input-Residual layer.

A configuration employing 320 threads per block (one thread per activation) was also evaluated; no performance improvement was observed, likely due to increased scheduling and synchronization overhead offsetting any benefits of finer-grained parallelism.

B. Parallelize Torso Residual Layer

To implement an efficient parallelization strategy for the torso residual layer, we designed the kernel based on its structure described in the previous section as follows.

To begin with, each kernel call (thread block/SM) is responsible for computing one output channel. To reduce the time spent on accessing data in the main memory, first the input channels are loaded into the shared memory of size 256 x 32 and the filters are loaded into the shared memory of size 36 x 32 in a way to mitigate bank conflicts. Specifically, each 8 x 8 input channel is stored in row-major order across one bank: the 64 elements of input channel 0 are stored in shared memory locations [0][0] to [63][0], the 64 elements of input channel 1 in [0][1] to [63][1], and so on. Once the first 32 channels are filled, input channel 32 is stored starting from [64][0], continuing in the same pattern. The same applies to the 3 x 3 filters. The total size occupied by the input channels and filters is 36.5 kB, which fits in the shared memory of our machine (64 kB).

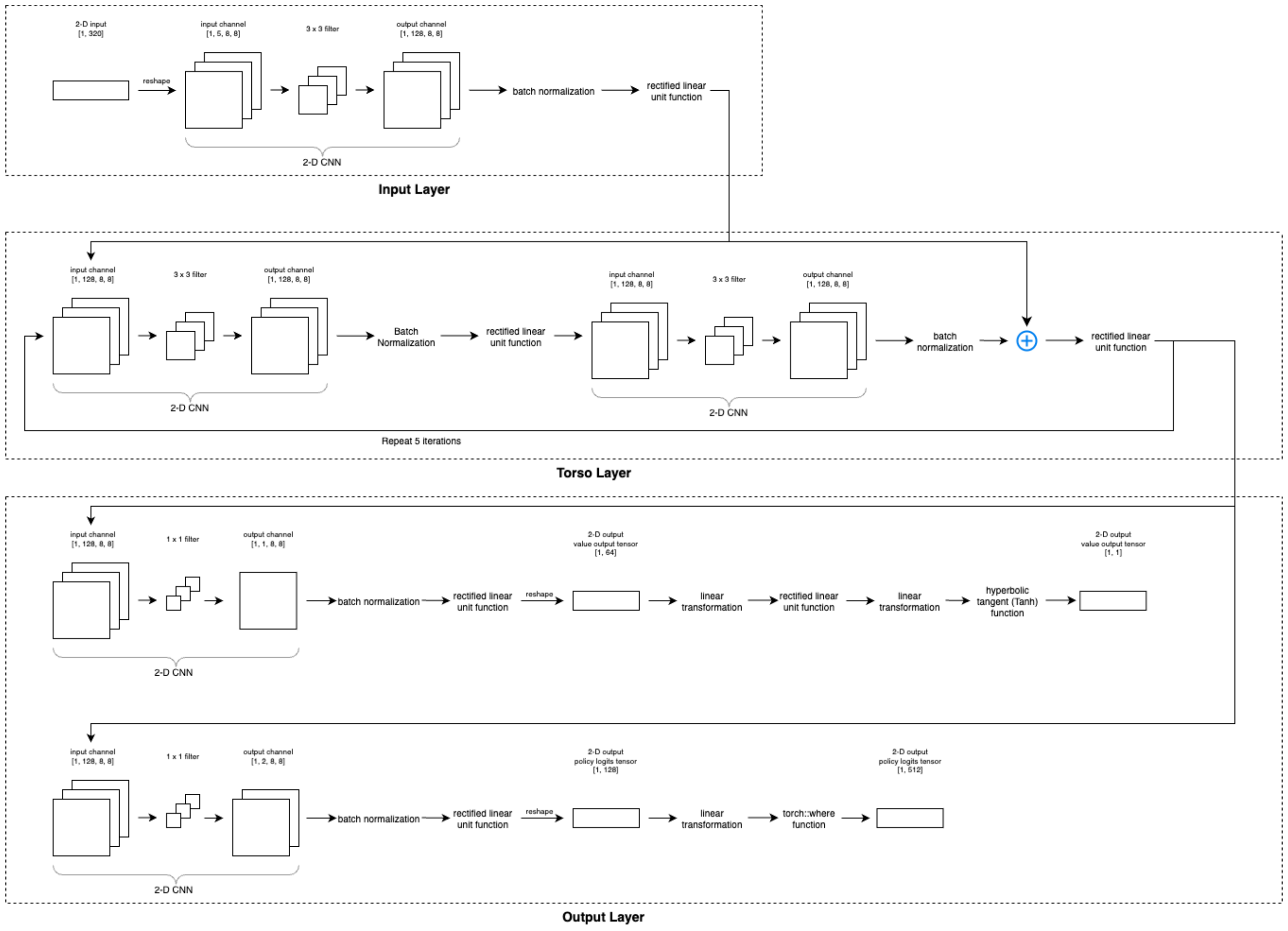


Fig. 1. Architecture of Convolutional Layer used in AlphaZero for Checkers

Each thread block is configured with 128 x 8 threads. Inside each block, every 128 threads form a group that computes one column of the output channel. Within each group, each thread is assigned to one input channel and its corresponding filter. For instance, thread 0 processes input channel 0 with filter 0, thread 1 processes input channel 1 with filter 1, and so forth. As a result, each thread in a warp consisting of 32 threads only needs to access one specific bank in the shared memory, without conflicts between each other. Each thread performs the convolution operation independently and utilizes atomic add to update the corresponding element in the output channel to prevent conflicts and race conditions.

Another design with a slight variation is to allocate shared memory to temporarily store the outputs and copy them back to the main memory after all computations are completed. This approach is feasible because the output tensor consists of a single 8×8 channel, occupying only 128 bytes. our experiments show that this strategy significantly degrades performance, because multiple threads within the same warp will attempt to write to the same element in the shared memory, leading to severe bank conflicts.

C. *Parallelize Output Residual Layer*

The output residual layer adopts the same parallelization strategy as the torso residual layer given that they share the same shape of inpit tensor, except that it only requires three kernel calls since there are only three output channels in total (1 in value CNN and 2 in policy CNN) and has a different filter size of 1 x 1.

Similarly, utilizing shared memory as a temporary storage for output tensors also decreases the performance in the output residual layer, as the bank conflicts remain a tremendous problem.

IV. Experiment and Results

Considering the scale and complexity of the AlphaZero source code repository, we extracted its convolutional process by writing a semi-program simulating its actual implementation in AlphaZero, to evaluate the correctness and performance of our parallelization strategy. After the pilot testing, we tried to integrate our customized convolutional kernels into the original repository.

A. Experiment Setup

The experiment is set up on the ECE Community Compute Clusters [3], specifically, on the machine ece022. Some of the machine hardware parameters are listed below:

- Model of the GPU: Tesla T4
- GPU Architecture: NVIDIA Turing
- Size of shared memory: 64kB
- Number of streaming multiprocessor (SMs): 40
- Number of CUDA cores (overall): 2560
- Total amount of GPU main memory available: 16GB

To conduct a thorough evaluation on the correctness and performance of our parallelization strategy, we first ran the three kernels which are pure CNNs with Libtorch built-in convolutional functions `torch::Conv2d` (short as ”kernel”), three wrapping layers (short as ”layer”) and the whole semi-model (short as ”model”) 20 times on both CPU and GPU respectively, to measure the baseline performance. Next, we ran the customized three kernels implementing our parallelization strategy, three wrapping layers and the semi-model 20 times on GPU, recording the performance and calculating the mean errors between our custom parallel implementation and the baseline on CPU and GPU.

Notably, the torso layer and output layer do not cover all functional units in the real architecture but only a part of it. The input, torso and output layers used in the experiment only consist of their most common part, which includes the CNN, batch normalization and ReLU.

B. Correctness Check

To verify that the custom CUDA kernels accurately replicate the behavior of libtorch's convolution routines, two complementary verification methodologies are employed: an element-wise tolerance check utilizing liborch's allclose function `torch::allclose` and a quantitative assessment via mean absolute error computation.

The first step involves a direct comparison between the output tensor generated by the CUDA kernels and the reference tensor from libtorch, employing an element-wise tolerance criterion. Specifically, for each corresponding element pair (x, y), the absolute difference $|x - y|$ must not exceed a fixed absolute tolerance ($atol = 10^{-5}$) plus a relative tolerance ($rtol = 10^{-8}$) scaled by $|y|$. Any violation of this bound signals the presence of indexing, accumulation, or numerical-stability errors within the CUDA implementation. The scaling of the criterion with respect to the reference value's magnitude ensures robust protection against unacceptable numerical drift across both small and large activation values.
Second, a global consistency metric is obtained by computing the mean of the absolute differences across all elements, resulting in a single mean-absolute-error value. Although a mean error on the order of 10^4 is generally regarded as acceptable for a single convolution—being significantly smaller than the typical scale of activation magnitudes—it must be emphasized that such discrepancies are not static. As feature maps propagate through batch-normalization, ReLU activations, and subsequent convolutional layers, the initial mean error is subject to transformation and amplification. Specifically, the batch-normalization process rescales and shifts activations according to batch statistics, while the piecewise-linear nature of the ReLU function can convert minor perturbations into persistent biases. As a result, a mean error of 10^4 observed after a single layer can increase by a factor of two or more following batch normalization, with further cumulative growth occurring across additional layers.

By systematically monitoring both the binary outcome of the element-wise tolerance check and the progression of mean-absolute-error following each major network block, it becomes feasible to detect and localize numerical drift introduced by the CUDA kernels before such discrepancies exert a material effect on final inference outcomes.

C. Performance

The average, maximum and minimum execution times of the kernels, layers and models presented in Fig 2, Fig 3 and 4 respectively. The computed mean speedups are shown in TABLE I.

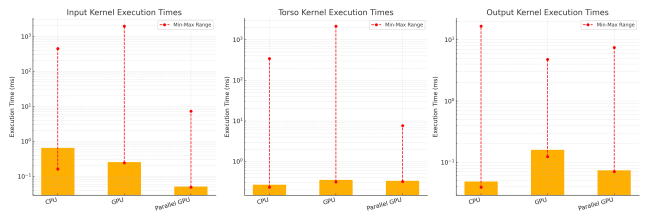


Fig. 2. Execution Times of Input, Torso and Output Kernels for checkers

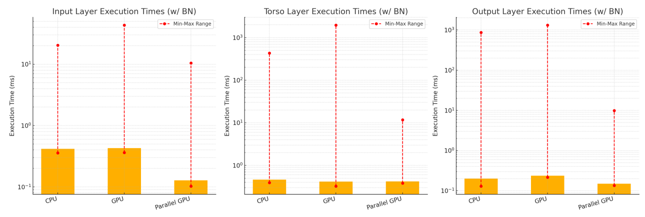


Fig. 3. Execution Times of Input, Torso and Output Layers for checkers

TABLE I
MEAN SPEEDUPS OF KERNELS, LAYERS, AND MODEL

Component	***Speedup vs GPU***	***Speedup vs CPU***
Input Kernel	4.968	12.739
Torso Kernel	1.051	0.795
Output Kernel	2.155	0.657
Input Layer	3.383	3.291
Torso Layer	0.991	1.109
Output Layer	1.593	1.382
Model	0.549	0.709

[a]Mean speedups are computed after excluding outliers (initial run).

The parallelization of the input kernel achieves the highest acceleration compared to the baseline CNN, with speedups of

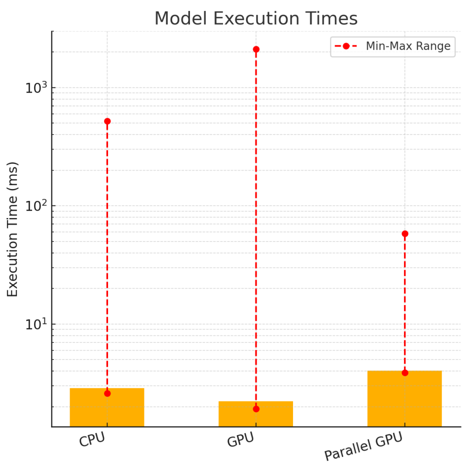


Fig. 4. Execution Times of Convolutional Semi-model for checkers

4.9× on CPU and 12.7× on GPU. The custom output kernel outperforms the baseline CNN only on GPU (2.2x), but is slower than the baseline on CPU. Meanwhile, the performance of the custom torso kernel is comparable to the baseline on GPU but worse on CPU.

The speedups generally drops when wrapping kernels into full layers, except in the case of the torso layer, where the speedup increases from ¡ 1 to 1.4.

It is notable that the baseline CNN consistently performs better on CPU than on GPU when running individual kernels and layers, indicating that in these cases the computational workload is relatively lightweight and the overhead from data transfers between CPU and GPU dominates execution time. In contrast, when running the full model, the computational workload is substantial enough to leverage the GPU's computational capabilities, effectively compensating for the data transfer overhead.

The semi-model incorporating all custom kernels underperforms the baseline on both CPU and GPU, despite the fact that the individual custom kernels and layers show better or similar performance. One potential explanation is that the semi-model with parallel implementations launches multiple kernel calls, resulting in additional overhead regarding scheduling and resource management. Another consideration is that the torso kernel w, which exhibits the worst performance among all kernels, constitutes the greatest portion of the model's total computation time, especially when it is made up of two CNNs and residual operations and needs to go through five iterations.

In addition, the maximum execution time for all cases consistently occurs during the first run, suggesting that the initialization is significantly more time-consuming. In this initial run, the execution time on GPU is usually noticeably higher than on CPU, as transferring data from CPU to GPU introduces additional overheads.

Furthermore, the execution time of baseline on GPU is significantly higher than our parallel implementation in the initial run. This is because Libtorch enables Deep Neural Network library (cuDNN) [4] by default, which performs extensive tuning to select the optimal algorithm for accelerating convolutional operations. In subsequent runs, the GPU leverages the optimal solution for the following computations identified by cuDNN, which outperforms our custom parallel strategy for the torso kernel.

The performance of the alternative parallel strategy for the torso and output kernels, which temporarily stores outputs in shared memory is illustrated in Fig 5. This strategy results in significantly worse performance for both kernels due to pronounced bank conflicts, caused by multiple threads within the same warp writing to the same shared memory element. As a consequence, this design is discarded as discussed in the previous section.

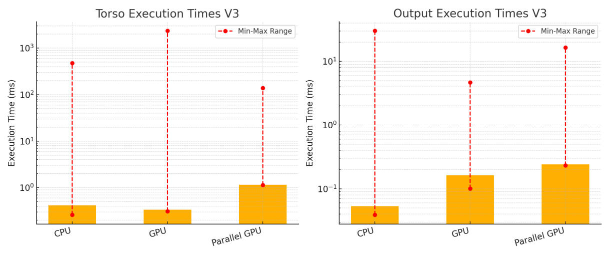
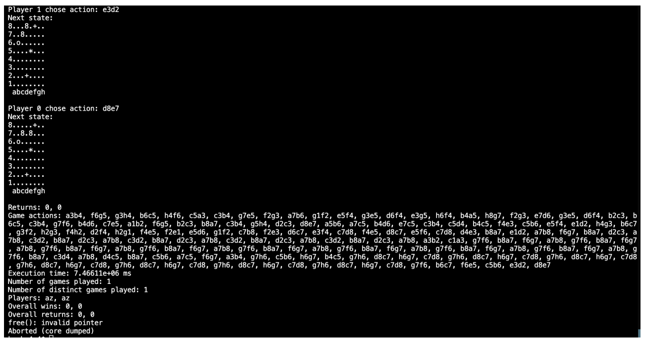


Fig. 5. Execution Times of Torso and Output Kernels for checkers with alternative parallel design

D. Integration

We have successfully integrated the three custom parallel CUDA kernels into the original AlphaZero repository and executed the game-playing inference. However, the program may abort at certain step because it fails to decide the next validate action based on the computed output, as shown in Fig 6. This issue arises because AlphaZero repeatedly runs the convolutional process and other computations thousands of times, during which floating-point errors accumulate. Eventually, the outputs fall outside the valid value ranges, leading to invalid actions.



```
Player 1 chose action: e3d2
Next state:
8...8.+..
7..8.....
6.o......
5....*...
4........
3........
2...+....
1........
 abcdefgh

Player 0 chose action: d8e7
Next state:
8.....+..
7..8.8...
6.o......
5....*...
4........
3........
2...+....
1........
 abcdefgh

Returns: 0, 0
Game actions: a3b4, f6g5, g3h4, b6c5, h4f6, c5a3, c3b4, g7e5, f2g3, a7b6, g1f2, e5f4, g3e5, d6f4, e3g5, h6f4, b4a5, h8g7, f2g3, e7d6, g3e5, d6f4, b2c3, b
6c5, c3b4, g7f6, b4d6, c7e5, a1b2, f6g5, b2c3, b8a7, c3b4, g5h4, d2c3, d8e7, a5b6, a7c5, b4d6, e7c5, c3b4, c5d4, b4c5, f4e3, c5b6, e5f4, e1d2, h4g3, b6c7
, g3f2, h2g3, f4h2, d2f4, h2g1, f4e5, f2e1, e5d6, g1f2, c7b8, f2e3, d6c7, e3f4, c7d8, f4e5, d8c7, e5f6, c7d8, d4e3, b8a7, e1d2, a7b8, f6g7, b8a7, d2c3, a
7b8, c3d2, b8a7, d2c3, a7b8, c3d2, b8a7, d2c3, a7b8, c3d2, b8a7, d2c3, a7b8, c3d2, b8a7, d2c3, a7b8, a3b2, c1a3, g7f6, b8a7, f6g7, a7b8, g7f6, b8a7, f6g7
, a7b8, g7f6, b8a7, f6g7, a7b8, g7f6, b8a7, f6g7, a7b8, g7f6, b8a7, f6g7, a7b8, g7f6, b8a7, f6g7, a7b8, g7f6, b8a7, f6g7, a7b8, g7f6, b8a7, f6g7, a7b8, g
7f6, b8a7, c3d4, a7b8, d4c5, b8a7, c5b6, a7c5, f6g7, a3b4, g7h6, c5b6, h6g7, b4c5, g7h6, d8c7, h6g7, c7d8, g7h6, d8c7, h6g7, c7d8, g7h6, d8c7, h6g7, c7d8
, g7h6, d8c7, h6g7, c7d8, g7h6, d8c7, h6g7, c7d8, g7h6, d8c7, h6g7, c7d8, g7h6, d8c7, h6g7, c7d8, g7f6, b6c7, f6e5, c5b6, e3d2, d8e7
Execution time: 7.46611e+06 ms
Number of games played: 1
Number of distinct games played: 1
Players: az, az
Overall wins: 0, 0
Overall returns: 0, 0
free(): invalid pointer
Aborted (core dumped)
```

Fig. 6. Screenshot of the AlphaZero inference for checkers interface

Addressing the floating-point rounding errors and re-integrating the custom CUDA kernels into AlphaZero will be left for future work.

V. Future Directions

There is still tremendous space to further improve the performance of our parallelization strategy. Some of the potential ideas are presented below.

- Reduce the number of kernel calls. Launching CUDA kernel calls introduces additional overhead due to resource management and synchronization. As a result, we might improve the performance by reducing the number of kernel calls required in the model, such as parallelizing the full model in one kernel call instead of computing each CNN in a separate kernel call.
- Consider using L2 cache. Only shared memory and main memory of GPU are used in our parallel implementation, while L2 cache exists in between these two of GPU's memory hierarchy. This needs further investigation of the use and features of L2 cache.

VI. Conclusion

In conclusion, we investigated into the architecture of AlphaZero's convolutional layer and laid a foundation for improving the inference by parallelizing the convolutional neural networks on GPU. We designed parallel strategies for three different convolutional kernels, conducted experiments, and evaluated the correctness and performance. We look forward to further improving the performance using more advanced parallelism design and mitigating the float rounding errors in the future work.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv preprint arXiv:1712.01815*, 2017. Available: https://arxiv.org/abs/1712.01815

[2] M. Lanctot, E. Lockhart, J.-B. Lespiau, V. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. De Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. Anthony, E. Hughes, I. Danihelka, and J. Ryan-Davis, "OpenSpiel: A Framework for Reinforcement Learning in Games," *arXiv preprint arXiv:1908.09453*, 2019. Available: https://arxiv.org/abs/1908.09453

[3] Carnegie Mellon University, *ECE Community Compute Clusters* [Online]. Available: https://cmu-enterprise.atlassian.net/wiki/spaces/ITS/pages/2332131370/ECE+Community+Compute+Clusters.

[4] Nvidia Developer, *NVIDIA cuDNN* [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/latest/