Parallelization of Expectimax Tree Search Based Temporal Difference Learning for the Game 2584 using GPUs

Wu Ho Lun Hsieh Ming En Tan Chien
吳赫倫 謝明恩 簡 聃
0316320 0316316 0316222
Computer Science Department, National Chiao Tung University, Taiwan
{allencat850502, twmicrosheep, whitetiger50512}@gmail.com

ABSTRACT

In this research, we explore the possibility of speeding up the training and inference time of expectimax tree search based temporal difference learning for the game 2584 by parallelizing the expectimax tree searching algorithm and parallelize the game playing process on multiple agents using multiple GPUs. We also created baseline methods using CPUs with OpenMP and C++11 threads for our reference. The parallelization of the tree searching process is able to speed up the inferencing time by more than 32 times and decrease the total training time needed by more than 62 times by utilizing two agents on two separate GPUs. The decrease of training time also means that we can explore deeper models in a more reasonable time which was previously impossible. Our deepest agent reached a max score of 212741, an average score of more than 66300, with the 2584 tile win rate of more than 86% with only three sets of simple 4-tuple features using the TD learning algorithm.

Keywords

Game 2584; 2584 AI; GPU parallelization; CUDA; Expectimax tree search; Game parallelization; Multi-agent learning; Temporal Difference learning; Position evaluation function; Reinforcement learning; N-tuple networks

1. INTRODUCTION

2584 is a game that is similar to the famous 2048 game. Instead of trying to combine blocks of f(t) and f(t) to form a f(t+1) bock, it combines blocks of f(t) and f(t+1) to form a f(t+2) block. This makes the game harder by adding the dilemma when a row of f(k), f(k+1), f(k+2) blocks presents sequentially. It becomes a choice between letting the f(k) and f(k+1) blocks combine thus changes the board to f(k+2) and f(k+2) or letting the f(k+1) and f(k+2) blocks combine to form a board of f(k) and f(k+3). This leads to tougher choices due to differences in distribution of the tiles on the board when different movement is chosen.

Temporal difference learning or TD learning for short, combined with tree searching algorithms can form state-of-the-art AI bots on the task of playing the 2584 game. TD learning requires a lot of games being played to train the different weights in the model. This process can take up to 200 hours for a 1 thousand game training run even when the numbers of layer searched is limited to 3. The most time consuming part of the current process is using the tree searching method to prevent local maximum choices. The huge amount of computation needed for tree searching makes it a perfect target for parallelization. Also, when training on GPUs, the ability to update the model on the device is important because large data transfer due to model updates is not efficient. TD learning combined with N-tuple networks can create models that can well fit inside the memory of the GPUs.

In this research, we implemented parallelization methods to speed up the training and inference process of expectimax tree search based temporal difference learning for the game 2584. We will first properly define our problem, which will be mainly divided into three sections, one for the game, one for the

parallelization process, and one for evaluation metrics. The first section will include the mechanism of game 2584, temporal difference learning, expectimax tree search and our game playing strategy. In the second section we will mainly talk about tree parallelization and game parallelization. In the last section, we will talk about the evaluation metrics that helps us compare the different methods we tried.

After properly defining our problem, we will follow on to propose our baseline models which includes single threaded CPU, tree parallelization using OpenMP task, two-level tree parallelization using C++11 thread, and CPU game parallelization. We propose these baseline models to help us compare the performance of our GPU methods.

With baseline models in place, we will continue on to describe the methods for GPUs, which includes tree parallelization using GPU, parallelization of temporal difference learning weight updates on the GPU, and game parallelization using multiple GPUs with different agents.

We will then discuss through the effectiveness of expectimax tree search in TD learning, comparing game and tree parallelization on both the CPU and GPU, its impact for the time needed on doing inferencing and training, how deeper models help us achieve new high scores, lockless weight adjustments and the various tradeoffs with the different aspects of GPU code.

2. PROBLEM DEFINITION

2.1 Game 2584

Game 2584 is a game with an N * N board which has N * N tiles and each tile is either empty or contains a number of the Fibonacci Series (Fib). In our research, we discuss the 2584 game with N=4. This game is played by two agents, Player and Evil. The game starts with two random tiles on the board. In each round, Player must move the tiles non-trivially in one of the directions, including upward, downward, left and right. We call the moving "action". The action must change the state of the board else it is illegal. If no legal moves are left, the game is considered finish. After the action is done, if a tile with Fib(x) meets a tile with Fib(x+1), they will be merged into a single tile with Fib(x+2) and Player will gain Fib(x+2) points. For example, 1 and 1 will be merged into 2, and 1 and 2 will be merged into 3. Moreover, the goal of Player is to get the highest points instead of staying alive.

For Evil, after each Player action, it chooses an empty tile to put a 1 or 3 with the probability of 75% and 25% respectively. The goal of Evil is to let the Player get the lowest points rather than reaching the game end. However, to simplify our discussion, the Evil in our research chooses the empty tile randomly.

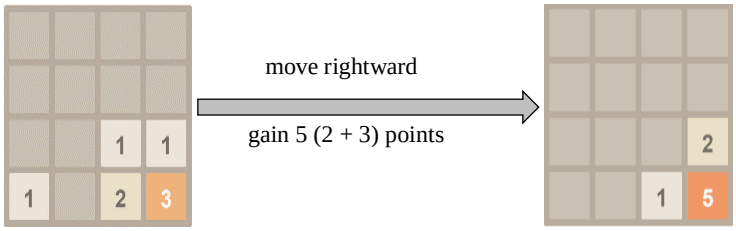


Figure 1. example game action for game 2584.

2.2 Temporal difference (TD) learning

Temporal difference (TD) learning is a prediction-based machine learning method. It has primarily been used for the reinforcement learning problem, and is said to be a combination of Monte Carlo and dynamic programming ideas. To discuss this learning method, we first define some terms related to ***Agent Environments*** as the following.

Agent Environments

- ***time step***
  - The round number which we collect data.
  - 0, 1, 2, …, T (T is the terminal time step)
- ***environment***
  - The state of the game at a specific time step.
  - $s_0, s_1, s_2, \dots, s_T$ (s_t is the environment at time step t)
- ***action***
  - How agent change the environment at specific time step.
  - $a_0, a_1, a_2, \dots, a_{T-1}$ (a_t is the action at time step t)
  - s_{i+1} is s_i changed by a_i
- ***reward***
  - The benefit agent gain after specific action is done.
  - $r_0, r_1, r_2, \dots, r_{T-1}$ (r_t is the reward at time step t, same as the reward of s_t after a_t is done)

With the ***Agent Environments*** well described, we go on with the state-value function, the most important part of TD learning. First, we define the total reward after time step t as the following.

$$R_t = \sum_{i=t}^{T} r_i$$

The ***state-value function*** $V(s_t)$ is defined as the expected total reward from s_t to the terminal environment (s_T), and the error

$$\delta_t = R_t - V(s_t)$$

The goal of TD learning is to minimize the error value of δ_t, so we need to adjust the value $V(s_t)$ by δ_t and a learning rate of α.

$$V(s_t) \leftarrow V(s_t) + \alpha * \delta_t = V(s_t) + \alpha * \left(R_t - V(s_t)\right)$$

What's more, we can define state-value function after the action as

$$Q(s, a) = V(s \text{ after } a \text{ is done})$$

In order to decide the action a_t at environment s_i, we can let

$$a_t = argmax_{a \in \text{possible actions of } s_t}(Q(s_t, a)).$$

2.3 N-Tuple Network

To calculate state-value function efficiently, we choose ***N-Tuple Network*** as the weight tables of the state-value function. In the N-Tuple Network, the tuples with different shapes and values of the tiles have different weights. We define

$\varphi(s)$ = tuples (features) list of environment s

$w(f)$= weight of tuple (feature) f

From the above we can obtain which can be used in the TD learning process.

$$V(s) = \sum_{f \in \varphi(s)} w(f)$$

2.4 Expectimax tree search

In traditional approaches, Mini-max tree search is usually implemented to obtain the best action at specific environment (state). However, since we consider that the opponent agent doesn't think in our thoughts, we select expectimax tree search algorithm instead. In this algorithm, the max value is chosen in our nodes while chance value (expected value) is used in the opponent nodes. Due to random choices of Evil's placement in our research, the expected value will be the averaged value in the chance nodes.

Not all nodes in the expectimax tree is valid, if the node is invalid due to rules of the game or if a terminal node is reached, the evaluation function is used to evaluate the environment at the terminal node, which would be an approximation of the relative reward compare to other terminal state. An expectimax search tree is illustrated in Figure 2.

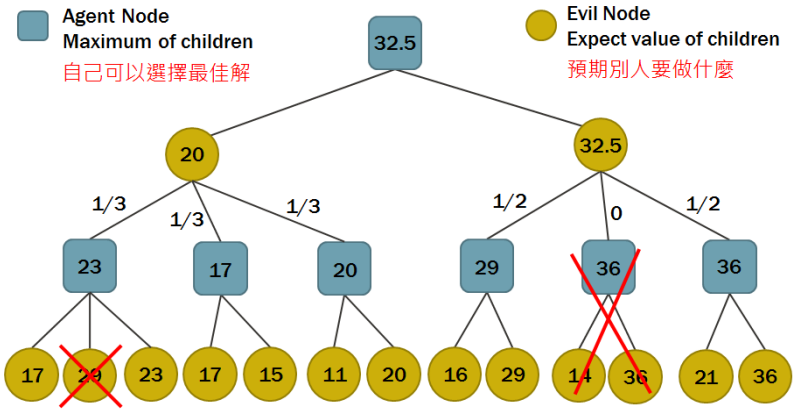


Figure 2. Example of expectimax search tree

To limit the search depth, the evaluation function is also called when the search process reach the max depth constraint. The evaluation function shall estimate the return value of going down that specific path. This greatly shortens the time of the search as many games such as 2584 may have more than 1000 steps, with a branching factor of 128 (4 directions * 16 locations * 2 possible tiles for evil), a full search of the tree is nearly impossible.

```
function expectimax(s):
    if s is terminal node:
        return evaluation(s)
    if s is max node:
        return max_{s'∈successors(s)}(expectimax(s'))
    if s is chance mode:
        return avg_{s'∈successors(s)}(expectimax(s'))
```

Figure 3. Psuedo code of expectimax tree search

To make it easier to describe the tree search process further down, we will group one set of evil tile placement and player movement as a "layer". We define an X-layer expectimax tree search to be an expectimax tree search with the max depth constraint set to 2X+1. Each layer consists of an evil and a player sub-layer, with an additional player sub-layer at the top for the decision making of the next step. A 0-layer expectimax tree search is going to search one player sub-layer and choose the action according to the evaluated value of the nodes after the action taken. An illustration of expectimax search tree for 2584 is illustrated in Figure 5.

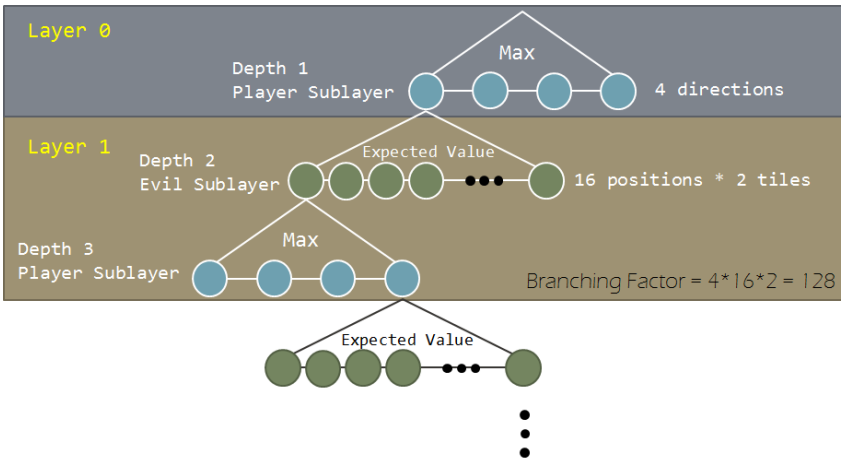


Figure 5. Example of expectimax search tree in 2584

In our research, we will mainly focus on 2-layer and 3-layer expectimax tree search. Some basic testing on 0-layer and 1-layer expectimax tree search is also done for comparison and discussion purpose.

2.5 Game Playing Strategy

In our research, we choose 4-Tuple Networks for TD learning, because weight tables of 4-Tuple Network don't consume memory resources exceeding the GPU's internal memory, while having well enough performance in learning. To further improve the performance of our approach, we combine TD learning and expectimax tree search to avoid trapping into a local optimized situation. To combine two algorithms, we choose the state-value function as the evaluation function of expectimax tree search.

We used 3 sets of 4-tuples in our model as shown in Figure 6. These 3 sets of features can form a solid model that performs close to other state-of-the-art approaches using 4-tuple networks. We will keep the tuple selection process simple in the research, since different tuple selection shall not impact the speed of the training and inference process too much.

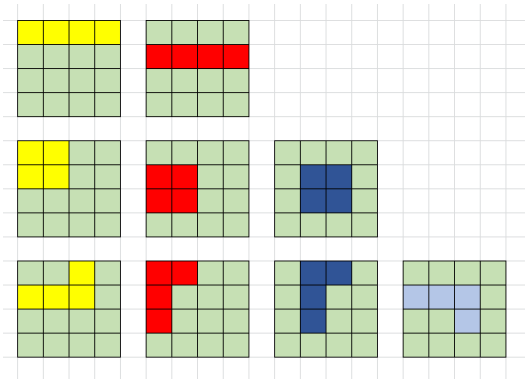


Figure 6. 4-tuples used in our research

2.6 Game History and Board Type

Different game boards may impact the speed of expectimax search. For boards that are at the beginning of the game, most nodes are valid, on the other hand, at the end of the game, many nodes are invalid. For recursive approaches and multi-level approaches, the amount of invalid nodes might influence the time in which the search takes.

To properly evaluate the influences of the number of invalid nodes, we define three types of boards through the history of the game. Each board has a history percentage according to its step in the game divided by the total steps in the game. For boards of history percentage 0 to 5, we call them starter boards. For boards of history percentage 40 to 60, we call them normal boards. For boards of history percentage 95 to 100, we call them late-game boards. Note that the agent in which we retrieve the history will also influence the classification of the boards we collected.

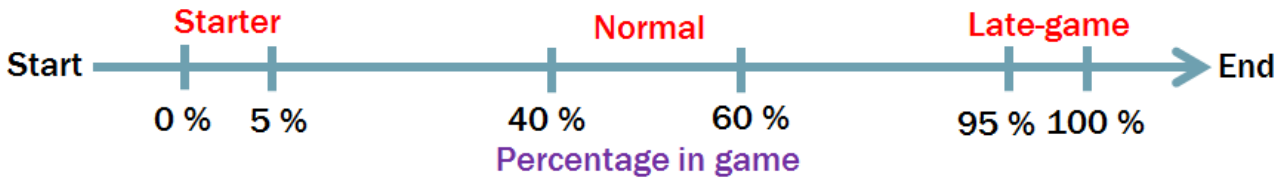


Figure 7. Game History and Board Type

3. RELATED WORK

Many algorithms were proposed to design AI programs for 2048-like games. Those algorithms often contain a tree searching procedure and an evaluation method to evaluate a state of the game board. For the game 2048, Szubert and Jaśkowski [1] proposed Temporal Difference (TD) learning together with N-tuple networks, to reach a hard to be beaten baseline model. Research [2] [3] also suggested that tree searching helps prevent the AI to choose the local best option and expectimax tree search can usually provide great results when there are chances in the occurrence of each node. Temporal Difference learning is used to train the weights of the N-tuple network. As proposed in [4], N-tuple network with expectimax tree search can be used to improve the performance of the AI.

The problem with tree searching is to find a balance between search time and the quality of search results. In games like 2584, it is impossible to search all the way to the end in reasonable amount of time. Most proposed methods focus on searching deep enough in the tree and use an evaluation function to evaluate the value of the state. Assuming the evaluation function is great, the results shall be the same as searching all the way to the end. In Veness's study [5], it was mentioned that an improvement that can be made to expectimax tree search is parallel search, since it enables the search to go deeper in the same amount of time.

As the training process goes, the AI agent will become better and better at playing the game, the length of each game will be longer and longer, thus slowing down the training process. Since in most conditions, the evaluation function is really fast comparing to the tree searching part, many attempts [6] [7] were made to parallelize the search tree process. Furthermore, some attempts [8] [9] [10] were made to parallelize the search tree process on the GPU.

As mentioned in [8] and [9], parallelization of tree search on GPU is different from the normal root-parallelism method on CPU. The difference is mainly due to batching of jobs and the execution scheme of the GPU. Recursion methods of searching will not perform as well on GPUs. Also, the communication cost between CPU and GPU is high, so the number of transfers needed to be minimized, larger chunks of data shall be processed at once and the tasks shall be in a high work to data ratio.

4. PROPOSED SOLUTIONS

We divide the parallelization process into six main steps: CPU Baseline, CPU Tree Search Parallelization Baseline, CPU Game Parallelization Baseline, GPU Tree Search parallelization, Parallelization Temporal Difference weight updating on GPU, and GPU Game Parallelization using multiple GPUs.

4.1 CPU Baseline

We first describe the version without GPU parallelization. The training process is as the following. The Evil agent first initializes the game board with 2 random tiles and delivers the board to the Judge. The Judge then alternates the board between the Player and Evil agent and checks if the game has end each turn. The Player will perform expectimax tree search on each round using the tuple networks weight as evaluation functions at the leaves of the tree. After each game ends, the history of the game is used by the Player agent to improve the N-tuple network weights using Temporal Difference learning.

To speed up the process, we change all boards into bit-boards. Consider we will only play to the 32 item in the sequence; we can use 5 bits to represent a tile on the board. With 16 tiles and 5 bit per tile, we can use __int128 type on the CPU to make operations faster. We then change all actions into corresponding bitwise operations, including player movement, evil action, rotation and reflection of the board. We will use recursion to traverse the tree.

4.2 CPU Expectimax Tree Search Parallelization Baseline

We start by setting a CPU tree search parallelization baseline. To achieve tree parallelization on the CPU, we separate the whole process into two-levels the upper level will be the usual DFS recursive process and the lower level will be split into parallel into many threads. We use OpenMP task and C++11 threads to parallel the lower level.

The location that separate the two layers is important as if too little task is split, the amount of IDLE threads will be large, many other threads might need to wait for the last or slowest thread to finish. If the location is low, too many tasks will be spawn and the overhead of putting a task into another thread and merging the result back will be large, resulting in performance and efficiency decrease.

4.3 CPU Game Parallelization Baseline

To speed up the training process, one way to do it is to decrease the training time as we did in tree search parallelization; the other way is to utilize multiple agents training at the same time. Considering 1000~2000 steps per game, 32^4 hash values per 4-tuple. If we use a lockless design, the collision rate is only about 0.1%, which is acceptable. We used C++11 threads to open multiple agents playing the same game and let them share the same TD table, when an agent finishes a game, it updates the shared TD table, then continue on to start another game.

4.4 Expectimax Tree Search Parallelization on GPU

To implement our game on GPU, we need to resolve several issues. First of all, we store our board in CPU intrinsic __int128, which is able to store a 128-bit integer, so that we can use bitwise operations to achieve several operations, such as player movement, evil movement, rotation and reflection of the board. However, there is no intrinsic int128 type on the GPU, so we store a board in an array of four uchar4. uchar4 is one of the vector types supported by CUDA library. The vectorized memory access is optimized by the CUDA library, so faster memory copy and access can be achieved.

To utilize the enormous computing power of the GPU, we have to have different approaches compared to the CPU. There are several difficulties while implementing the same algorithm on GPU. For instance, recursion is not easy to implement on the GPU if we want optimal performance. Another problem is that GPU runs the same piece of code in parallel with thread amount that is thousands of times larger than the CPU. Too many thread synchronization will cost a lot in GPU performance thus slows down the total throughput.

As a result, we split the complex and trivial computations into two major parts. One is the generation of all necessary values on the search tree such as board, current value and current status. And the other is

the reduction of values on the search tree, which reduces the successor's status into a single value. For generation, we will generate all the possible paths and generate the search tree from root to leaf nodes. And for reduction, we want to mimic the tree search process and merge the results of the leaf nodes to their parent node until root node according to the values we store when we generate the search tree.

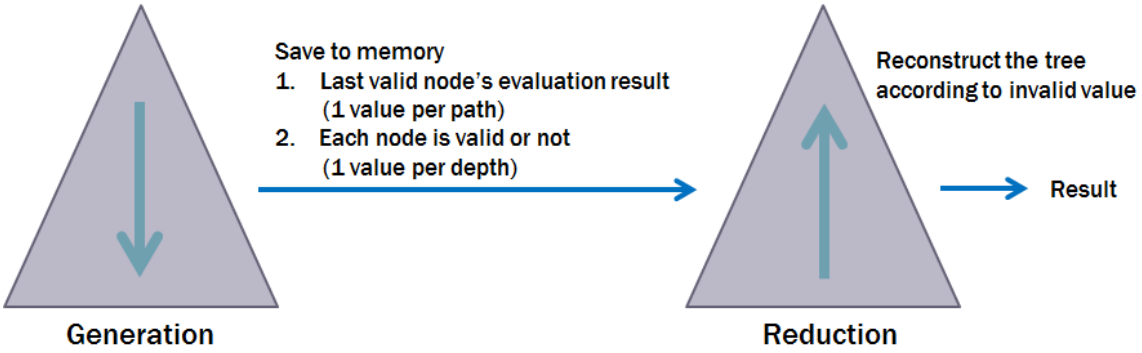


Figure 8. Generation and Reduction

Apart from the original method of using CPU, that is, recursive algorithm, there is an inevitable issue we will need to conquer when separating the algorithm into two parts. There will be invalid status in both the generation and reduction processes. For example, a board might not be able to put a tile or not able to move in a certain direction or even that the board is the end of the game. An invalid board status will result all its successor board status to be invalid. And when it comes to reduction, we will have to know whether to return the evaluation value or to get the max value of the successors or the average value of the successors to our parent node. For instance, when calculate the expected value; we can't use the evaluation value with invalid status to calculate the average since the nodes are not even possible to reach in reality.

To enable us to correctly reconstruct the tree in the reduction process, we will need to save minimal but enough information. Saving too much information will increase the time needed for memory operations. We will save two piece of information per path, the last valid node's evaluation result and N values indicating each depth is valid or not with N equal to the maximum search depth.

The process of generation can be categorized into two approaches. One is naïve approach and the other is multi-level approach. The naïve approach parallels all the possible moves and positions to put tiles. For example, a search tree with depth N, we will use a total of 128^N threads to parallelize the generation process. For each thread, it has a global id that corresponds to a particular path from the root node to the leaf node.

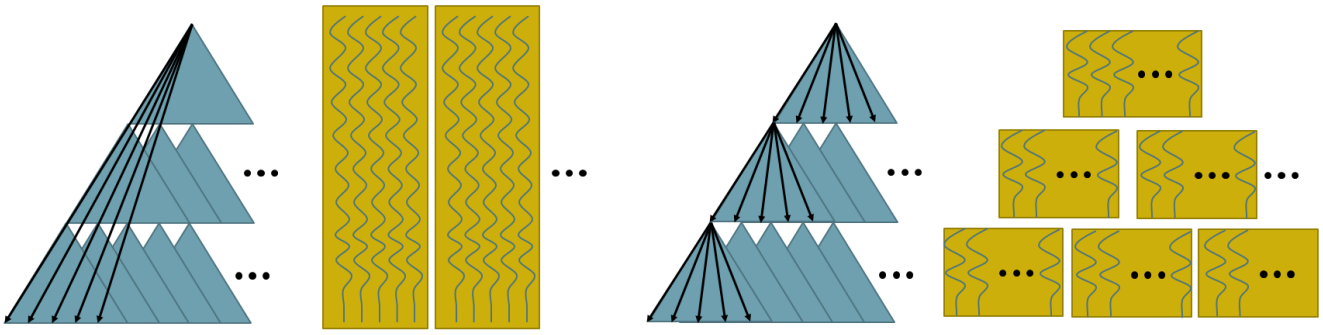


Figure 9. Naive Generation versus Multi-level Generation.

This is the most basic approach to implement it on the GPU; however there exists two optimization issues. The first issue is that most threads are doing the same work in the beginning with only slightly different behavior at the end. To give a more concrete concept, in a 3 layer search tree, there will be 128^3 threads doing the same operation with only having 128 different results in layer 0. When the layer count is small, this might be beneficial due to less kernel launches, but as the layer count increases, this amount grows exponentially, which will become an inevitable loss in performance. The other issue is that early break can happen frequently in late-game boards. Take a board that can't be moved up in the first move as an example, it still has to run through all the 128^2 successors that is simply invalid. Even worse, if all but one thread is valid in that thread block, all other threads still needs to wait for it to finish the traversal, which can hit the performance hard considering the amount of IDLE threads it introduced. With late game boards that die pretty quickly, this can contribute to a huge performance loss.

Algorithm 1

```
1: procedure GENERATE(depth)
2:     if parentMoveValid is VALID then
3:         if parentBoard position (threadRow, threadColumn) is empty then
4:             currentBoard(threadRow, threadColumn) := threadTile
5:             if currentBoard can move in threadDir then
6:                 reward := currentBoard.move(threadDir)
7:                 threadBoard := movedBoard
8:                 threadEvilValid := VALID
9:                 threadEvilValue := parentEvilValue
10:                if depth == maxDepth then
11:                    threadMoveValue := parentMoveValue + reward + evaluate(movedBoard)
12:                else
13:                    threadMoveValue := parentMoveValue + reward
14:                end if
15:            else
16:                threadEvilValid := VALID
17:                threadEvilValue := parentEvilValue + evaluate(currentBoard)
18:                threadMoveValid := INVALID
19:                threadMoveValue := 0
20:            end if
21:        else
22:            threadEvilValid := VALID
23:            threadEvilValue := 0
24:            threadMoveValid := INVALID
25:            threadMoveValue := 0
26:        end if
27:    else
28:        threadEvilValid := INVALID
29:        threadEvilValue := 0
30:        threadMoveValid := INVALID
31:        threadMoveValue := 0
32:    end if
33: end procedure
```

Therefore, we proposed a multi-level search hierarchy to enhance the performance. We break up each search by depth and do calculation one depth at a time. This solves the two problems mentioned previously by calculating the higher levels only once and skipping lower layer generation when possible. However, as we divide the task in to smaller parts, there will be more overhead for kernel launches and memory operations. Each thread is still assigned with a global id that corresponds to a particular path from the parent node to the node of its successor. The pseudo code is shown as Algorithm 1.

The process of reduction is designed using a multi-level architecture. We first allocate a shared memory of size 128 for the data synchronization between 128 threads, which represents the 128 child nodes of a same parent node. Moving all data to shared memory greatly increased the performance since the access time increases by more than 20 times if we did not move to shared memory and use the global memory locations. The first step of reduction is to find the max value of the four directions. We will need $\log_2 4$ comparisons to perform this step. When performing each comparison, we also have to check the valid status of the target values to compare. If the target values are invalid, we will not consider its value. If all the values are invalid, we will use the evaluation of board without moving. We will store the intermediate results to the first 32 shared memory locations for later use. By merging the data to the first warp, we can utilize the fast memory access in same warp using the __shuf() command. We then calculate the average value of the 32 intermediate results. We will need $\log_2 32$ summations. Since 0 will not affect the result of average, we store 0 for the evaluate value and valid value during the generation process if that path is invalid. Therefore, we can sum up the values and count the valid thread of the thread easily. If the count is greater than 0, which means there must be at least 1 valid child, we will update the value of parent node with the summed value divided by valid count; otherwise we return the evaluated value of the board. By doing so, we are mimicking the bottom up reduction of the expectimax search tree. The pseudo code is shown below as Algorithm 2.

Algorithm 2

```
1: procedure REDUCE(depth)
2:     for i ← 1 to log2 4 do
3:         for all thread in parallel do
4:             threadMoveValue_k := max(threadMoveValue_k, threadMoveValue_{k+1})
5:             threadMoveValid_k |= threadMoveValid_{k+1}
6:         end for
7:     end for
8:     if threadId % 4 == 0 and threadMoveValid_k then
9:         threadEvilValid_{k/4} := threadMoveValid_k
10:        threadEvilValue_{k/4} := threadMoveValue_k
11:    end if
12:    count := threadEvilValid
13:    value := threadEvilValue
14:    for i ← 1 to log2 32 do
15:        for all thread in parallel do
16:            count += threadEvilValid_{k+(1<<i)}
17:            value += threadEvilValue_{k+(1<<i)}
18:        end for
19:    end for
20:    if threadId == 0 and count != 0 then
21:        parentMoveValue = value/count
22:    end if
23: end procedure
```

4.5 Temporal Difference weight updating on GPU

The second part of the parallelization process is to parallelize the process of updating the weight tables. We first evaluate the score of each board with the original weight tables, and then calculate the amount of penalty we want to update using the TD learning method mentioned previously, each thread is given a specific time step to calculate and update. Since we have a lot of threads in the GPU and all these weight updates calculations can run independently, we can greatly speed up the time needed to run the update. Furthermore, atomic operation provided by CUDA library guarantees that all adjustments of all features can be done in the same time correctly. Furthermore, updating the weights on the GPU can decrease a large amount of communication needed after each game is played, since we do not need to move the weight tables back and forth anymore.

4.6 GPU Game Parallelization using multiple GPUs

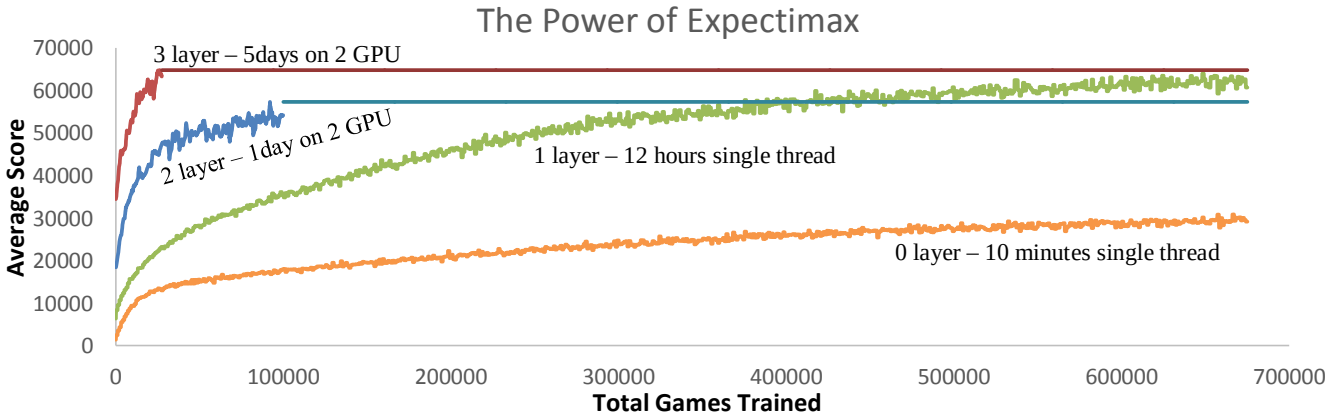
With multiple GPUs, we can further speed up our training process. Such as CPU game parallelization, we can start one game on each GPU, so that we only need ½ training time to achieve the same average score if we have two GPU. To implement multiple games on GPU, we launch one thread for each game on CPU to avoid blocking between different games. Besides, we create one stream for one GPU to avoid blocking between kernel launching on different GPUs. The default stream provided by CUDA library will block each kernel launching and CUDA API. We also use cudaMemcpyAsync() to copy memory on different GPU simultaneously. Finally, the most important part of multiple games is the correct update of TD weight tables. To update weight tables correctly, not only does each game update its own weight tables but also others' weight tables. So we copy the history boards and history rewards from CPU to multiple GPUs with their own stream simultaneously, and then update the TD weight tables on multiple GPUs concurrently. Due to FIFO execution of CUDA stream, a GPU only executes either expectimax tree search or TD weight tables update at one timestamp, which guarantees the correctness of the expectimax tree search.

5. EXPERIMENTAL METHODOLOGY

For our experiments, we use two different machine types. For CPU tests, we use AWS EC2 c4.8xlarge instances which have multiple Intel Xeon E5-2666v3 (2.60 GHz, 10C/20T) inside forming 36 virtual cores, with 60 GB of RAM. For our GPU tests, we used a dedicated server with an Intel Core i7-5820K (3.30 GHz, 6C/12T), 80 GB of DDR4 RAM and two NVIDIA GTX1080 8GB. For test terminology, see the previous problem definition section.

6. EXPERIMENTAL RESULTS

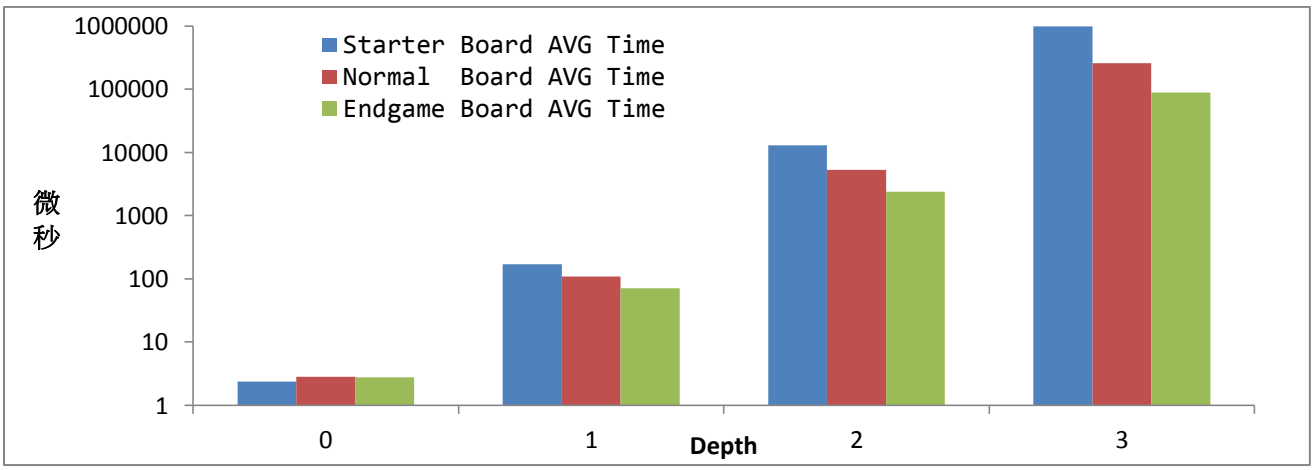
6.1 Effectiveness of Expectimax



From the graph we can see, as the number of layers increase, the average score of the agent increases dramatically. The average score of a 2 layer agent at start can win a 0 layer agent with full training easily. With only a few amount of games trained, a 3 layer agent can easily reach the maxed out score of a 1 layer agent.

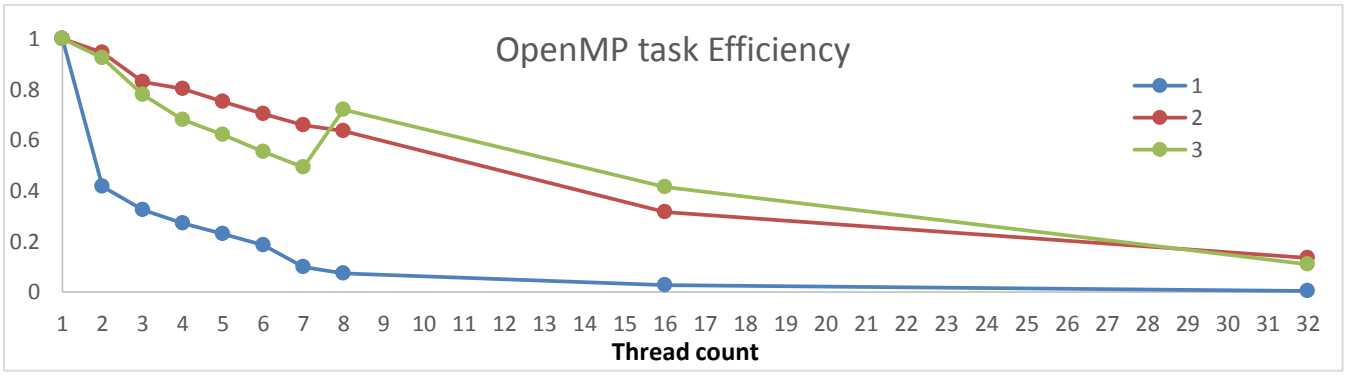
From this chart we can also see that parallelization is important if we want to utilize a deeper model. Even with two GPUs, it still takes a lot of time for the agent to train a small amount of games. To max out its performance, it will take a really long period of time without parallelization, which will make it infeasible.

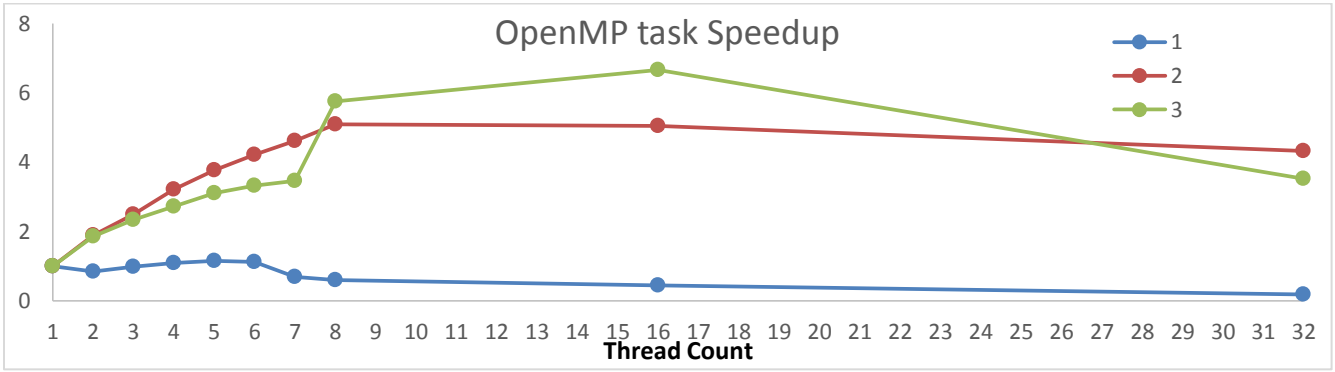
6.2 Game Board's Influence on CPU Baseline

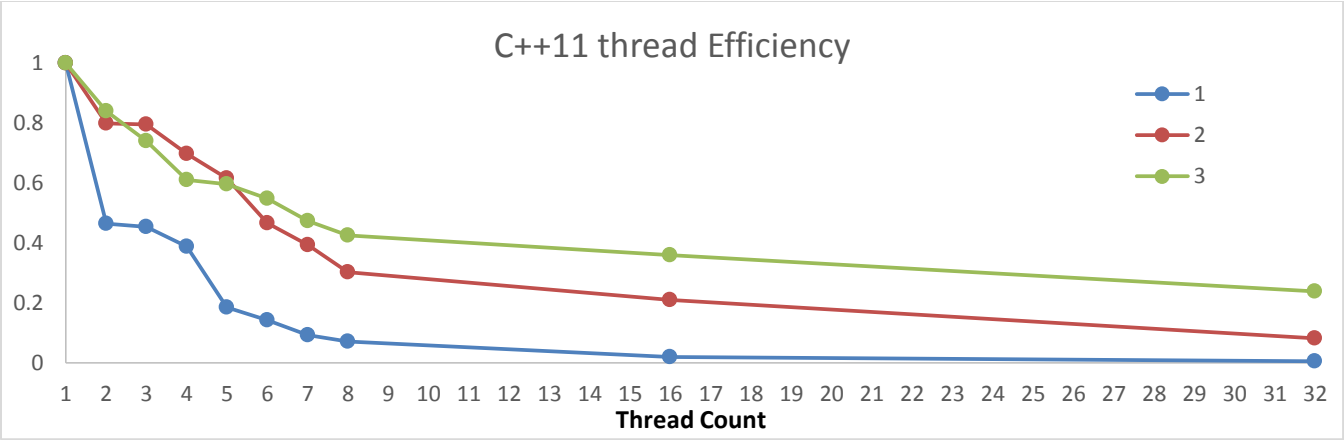


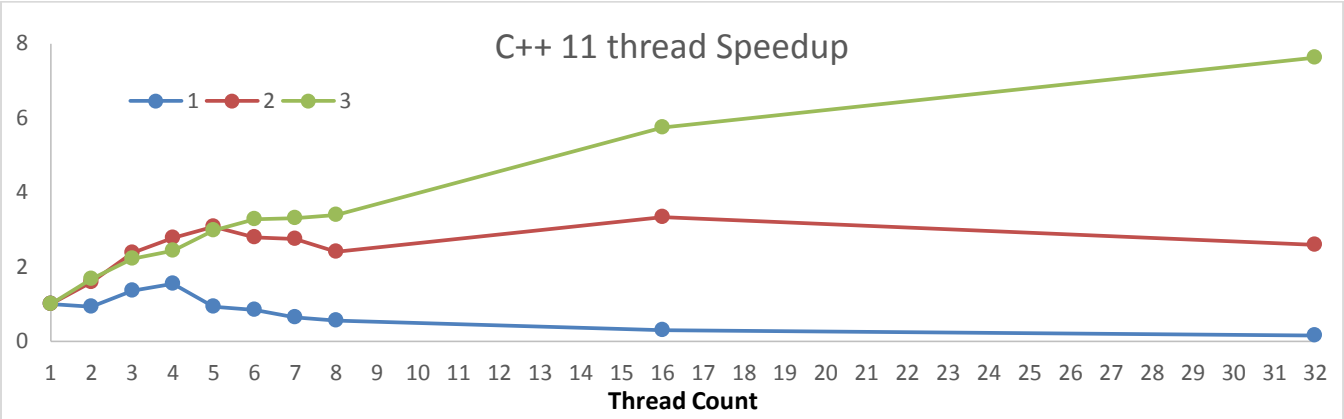
From this chart, we can observe the influence of different board types. The chart is changed into log scale for better visualization. We can observe that the deeper we search, the time we need increases. Also, the difference between end-game boards and starter boards increases as we search deeper.

6.3 CPU Tree Parallelization





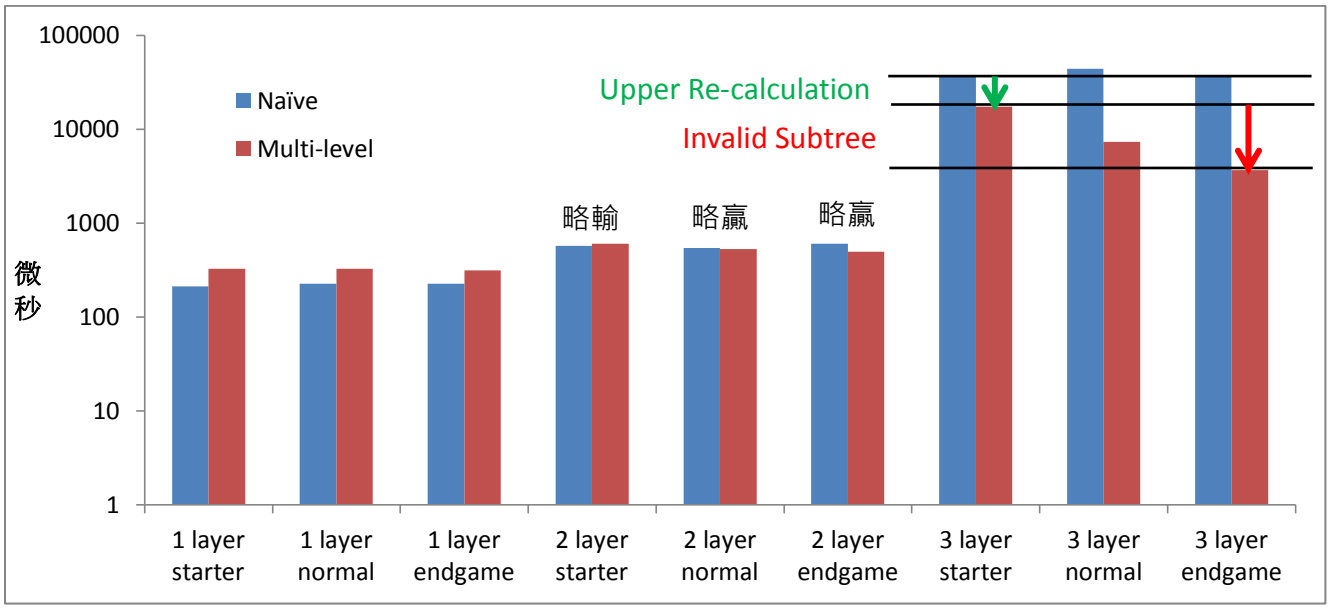




With large amount of threads, the work became too distributed, and the efficiency drops significantly due to the large amount of overhead. As we can see for some parameters, as threads increase from 16 to 32, due to a really low efficiency, the speedup even drops.

Although tree parallelization decrease inference time, but low efficiency means that more thread won't help and there's a maximum speed up.

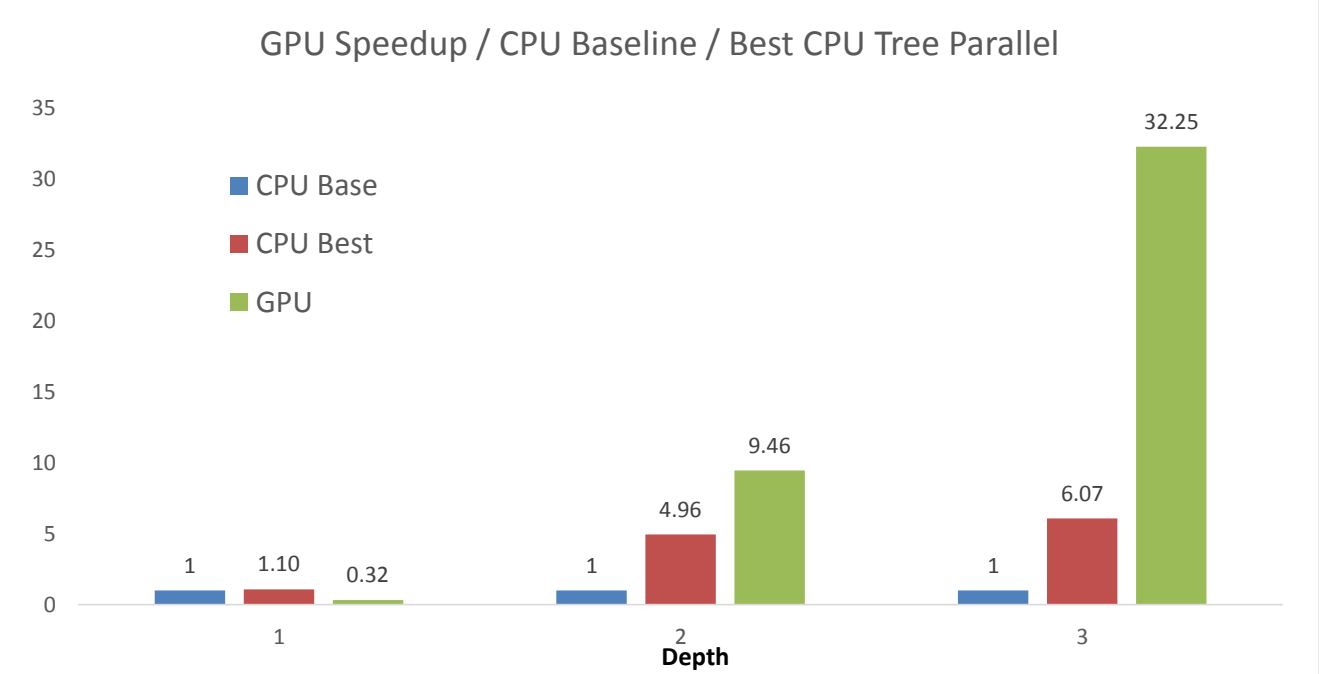
6.4 Tradeoff Between two GPU Generation Methods



From this chart, we can observe that when performing a 1 layer search, the Naïve method will be faster due to less overhead. As the layer number increased, the difference between the two increases, and the multi-level approach is the final winner.

Taking a closer look at the three layer timings, we can conclude the impact of upper layer re-calculation and invalid subtree. When evaluating using starter boards, most directions are walkable, the amount of invalid subtrees shall be smaller, so the time difference shall be mainly due to the reduction of upper layer re-calculation. When evaluating using end-game boards, lots of subtrees are invalid, so the reduction between the time consumed shall be due to the increase amount of invalid subtrees that it skips faster.

6.5 GPU Tree Parallelization Speedup



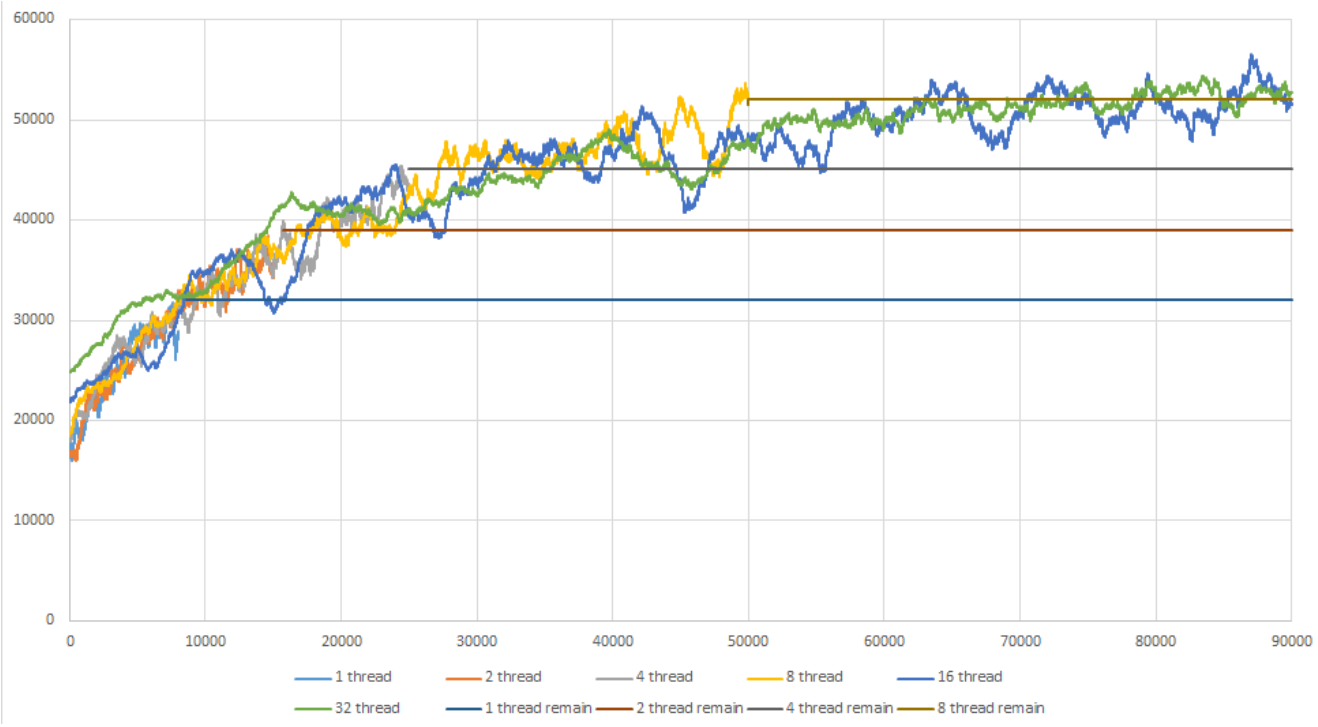
Comparing the CPU baseline with CPU tree parallelization's best speedup and a single GPU speedup, we can conclude that the difference between CPU parallelization and GPU parallelization will be larger and larger as the number of layers increased. At 1 layer, due to GPU overhead, the speedup will be less than one, so CPU parallelization will win in this case.

6.6 GPU Temporal Difference Weight Update Parallelization

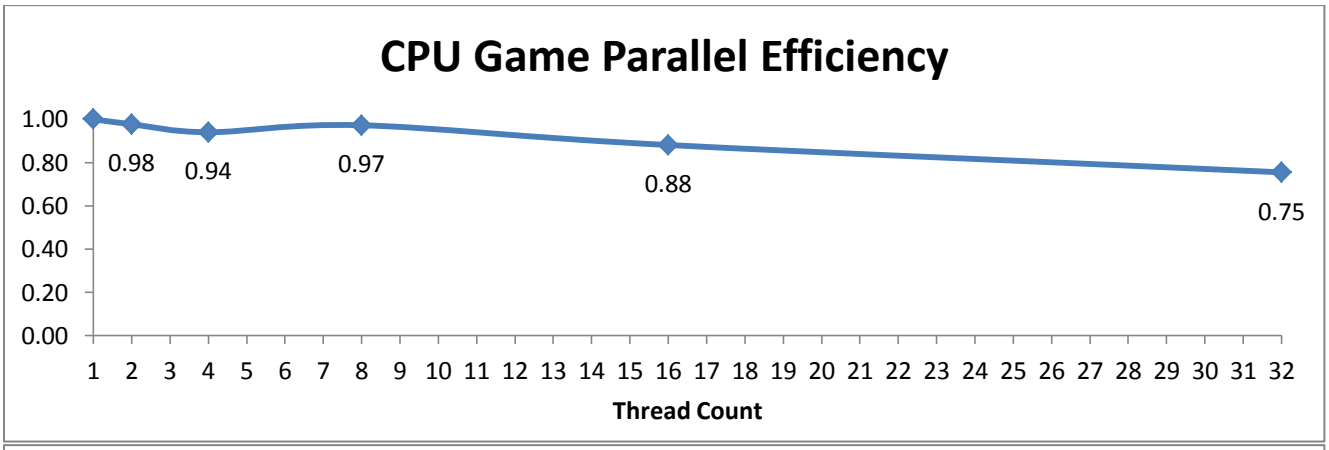
Comparing the time that the GPU takes compared to the CPU, we can see that by parallelizing the update of temporal difference weight on the GPU, we can obtain more than 12 times speedup.

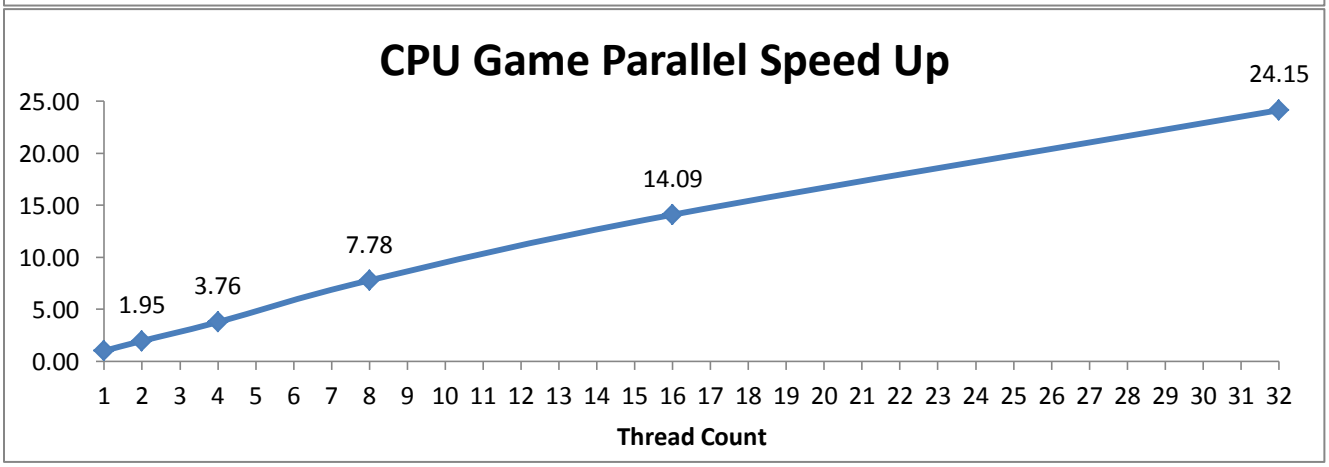
1000 Games 1000 Rounds/Games update	CPU	GPU	Speedup
Time (ms)	1197	95.80634	12.49

6.7 CPU Game Parallelization



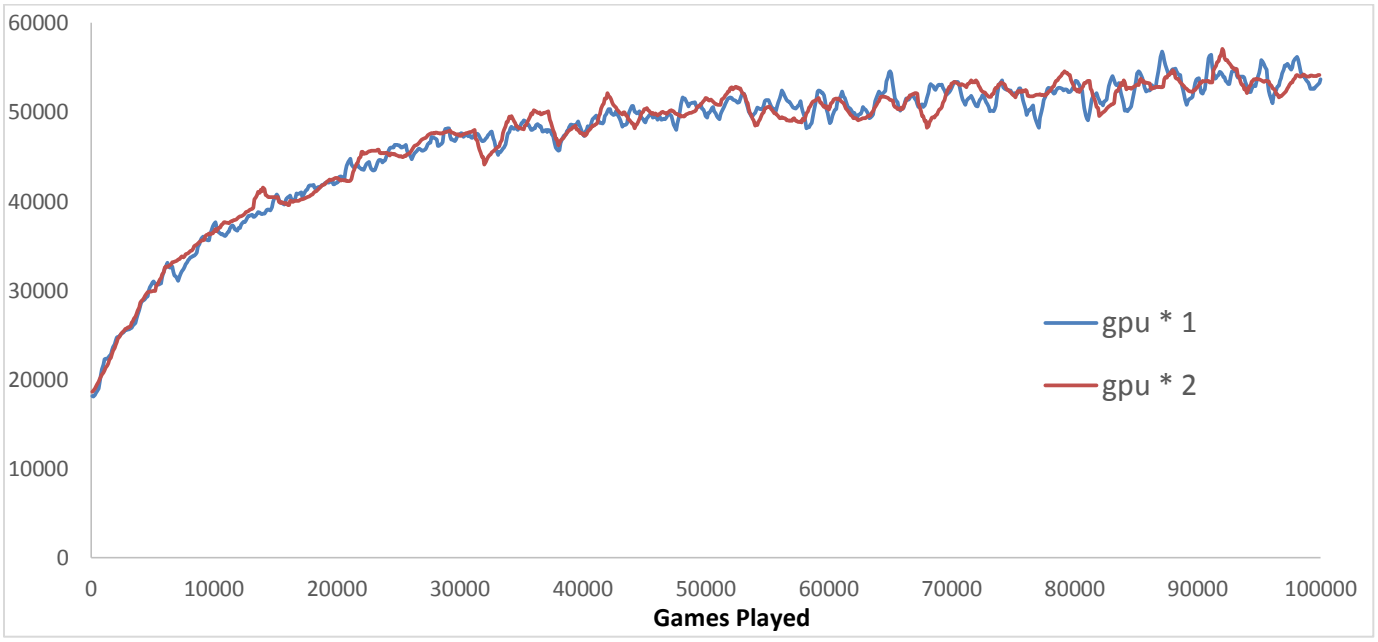
From this graph, we can see that although using a lockless design and having a 0.1% collision rate, it did not affect the learning of the agent, all agents climb in a similar way regarding the amount of threads.





The efficiency is greatly increased due to more compact workload, and the speedup grows to as much as 24 times when using 32 threads. Although game parallelization does not speedup inference time, it can greatly decrease training time.

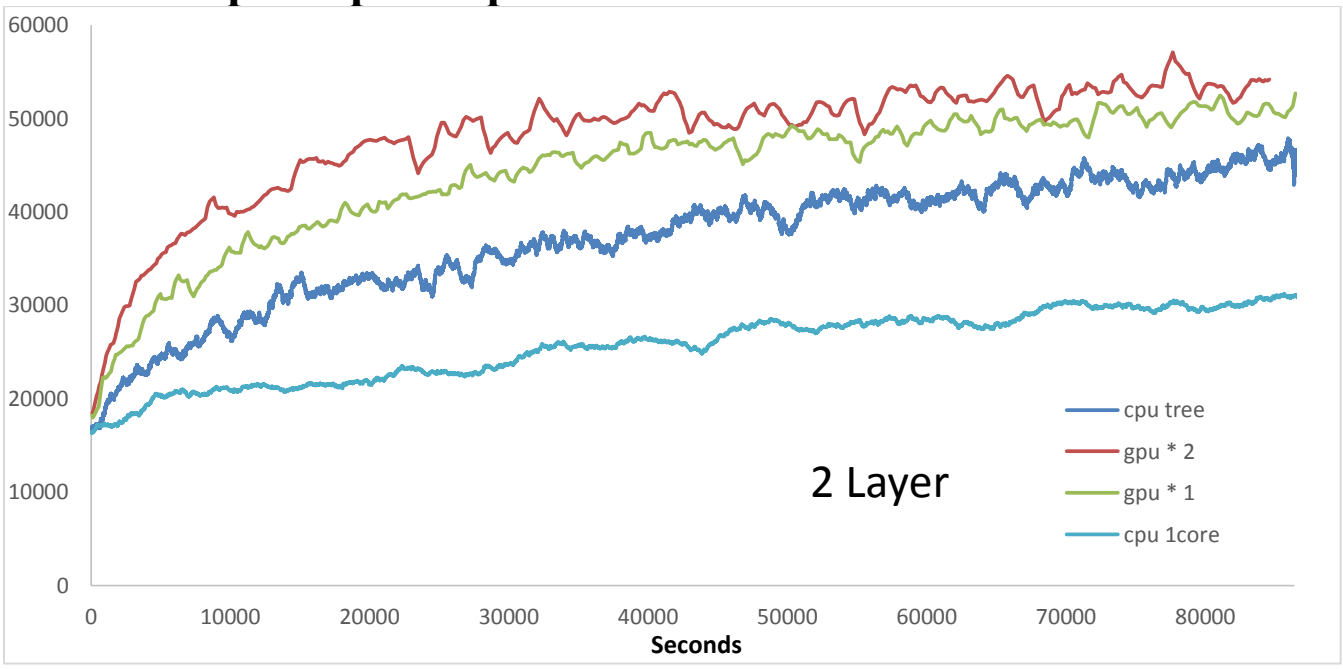
6.8 GPU Game Parallelization



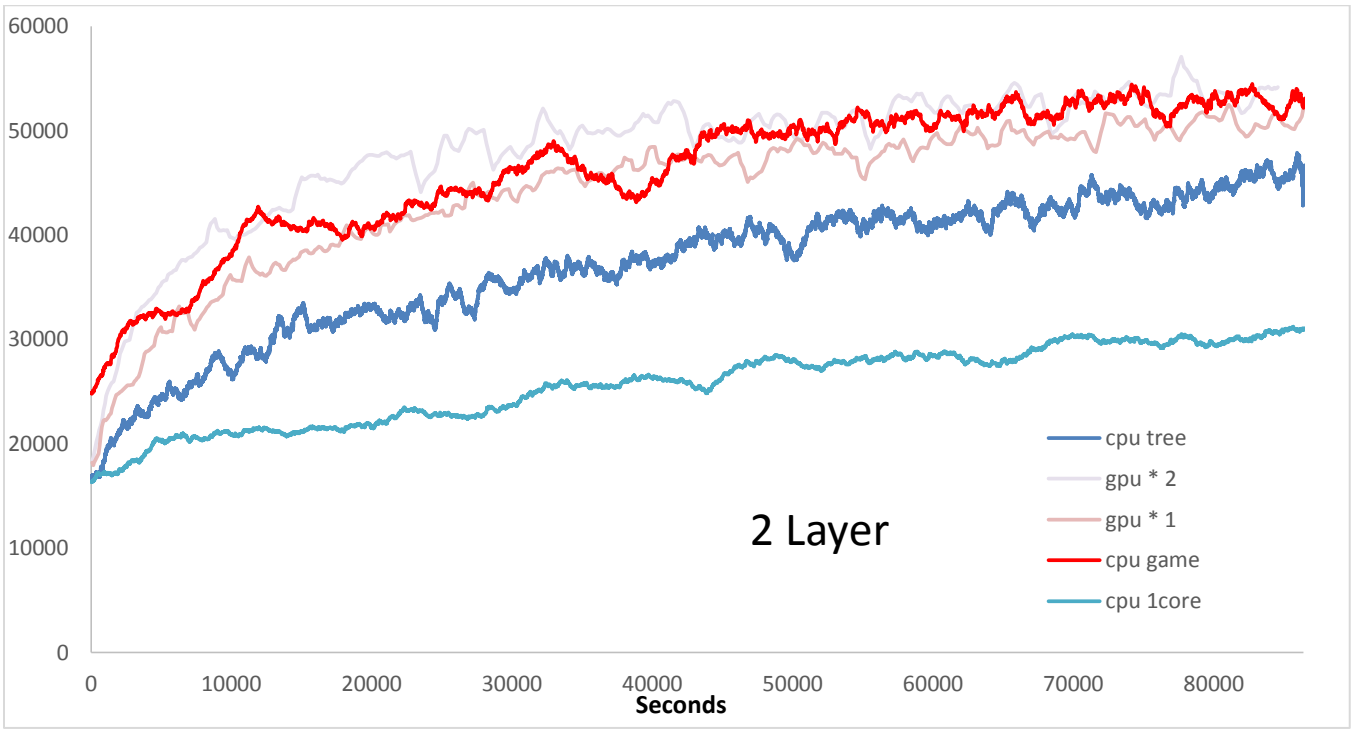
From this graph, we can confirm that using multiple GPUs will not lead to performance decrease.

	Efficiency	Speedup
2 layer	0.975	1.949
3 layer	0.973	1.945

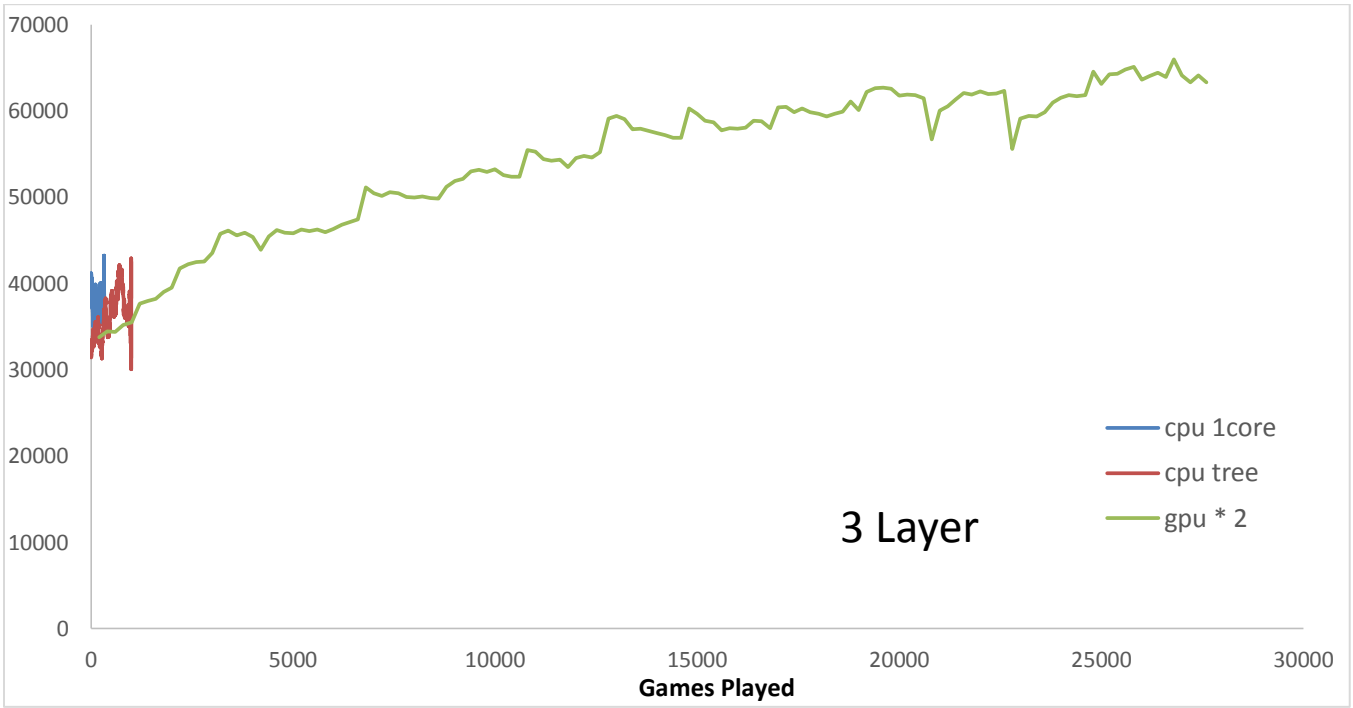
6.9 Total Speedup Comparison



From the graph above, we can see that the faster the training for one game, we can have a better performance with the same amount of time.



At 2 layers, a 32 thread CPU game parallel agent, can be close to the performance of 2 GPUs and at 3 layers, 2 GPUs is faster than a 32 thread CPU game parallel agent by 2.5 times.



At three layers, the CPU versions are just simple too slow to compare with the GPU. So we change to visualize it with the amount of games played.

7. DISCUSSIONS

When discussing the performance of an AI agent, there are two terminologies, inference time and training time that are often mentioned. Inference time is defined as the time consumption that the agent costs to give an output, after an input was received; however, training time is defined as the time consumption of the agent to achieve a specific level. It is common sense that the training time decreases along with the reduction of inference time, but training time can also be decreased by using multiple agents.

Considering the inference time, in 3-layer model, one GPU has more than 30 times speed up to our CPU baseline and more than 5 times performance increment than the best performance of CPU tree parallelism. However, when it comes to the training time, GPU still owns 30 times speed up than the CPU baseline, but game parallelization with 32 threads on CPU is able to reach the same performance of one GPU. However, a 32-core computer, which is incredibly expensive and hard to have for a normal user, has nearly the same performance with a computer equipped with one NVIDIA GTX1080.

Comparing parallelization of CPU and GPU techniques, we can see that when the depth of search tree is 1, CPU parallelized approach is faster and performs better than GPU approach. The reason is that when the search tree isn't deep enough, there's not much task to do and the overhead of creating GPU kernels, allocating memories, and copying data between device and computer is extremely huge. However, when the depth comes to 2, the overhead comparing to the benefit GPU bring is much lesser. Furthermore, when the depth of search tree exceeds 3, even a 32 core computer is even unable to train the game playing agent because it need too much time. On the other hand, a computer with 2 NVIDIA GTX1080 performs better and cost much less time instead.

According to the experiment results, we know that the performance of GPU parallelization is about three times slower than our CPU baseline in 1-layer model. To delve into the reasons, we use NVIDIA Visual Profiler to analyze out implementation and show the time cost by each step. After profiling, we notice that CUDA library spent more time in launching kernels, synchronizing and copy memory between host and device than computing. Thanks to the profiler, we realize that GPU parallelization has poor performance since there is not enough tasks for GPU and the overhead of CUDA library is much larger than computation.

When implementing game play parallelization on GPU, we made several optimizations. Since we have 2 GPUs, we created 2 CPU thread to handle GPU game plays in order to avoid blocking within game plays sharing same CPU thread. Also, it is not necessary to use lock to make sure the correctness of modification of weight table because GPU provides atomic operations. And the FIFO characteristic of streams makes sure that there will never be two kernels, updating weight tables or expectimax searching tree, working at the same time which ensures correctness of the agent.

There are several reasons that slow down a CUDA program. When we are struggling to optimize the performance, one of the reasons is branch divergence. Branch divergence happens when there is a branch instruction in same warp. In a warp, all threads share the same program counter, and therefore if a conditional branch out occurs, all the other threads have to wait and the while performance slows down. As a result, we made optimizations, including writing branchless code and using ternary operators on our implementation to have the least branch and reduce the scope of divergence in order to reduce performance decrement of branch divergence. After we made such modification, we have successfully reduced branch divergence drastically and get better performance.

Due to hardware restriction, the total amount of registers of a single block in GPU is limited. In other words, the number registers used in single thread will affect the number of threads of a single block. The SM of GPU executes a block at a time. Thus, if there is not enough warps for the scheduler to schedule,

the SM will be in idle state. On the other hand, there might be operations causing low active warps, for example, __syncthreads(). As a result, we have to carefully design the size of blocks, registers used in single thread, and the synchronization to optimize the throughput.

8. CONCLUSIONS

In conclusion, we have done parallelization on both CPU and GPU, setting concrete baselines using CPU parallelization. For inferencing, GPU can speed up more than 32.25 times compare to CPU Baseline and can speed up more than 5.31 times compare to CPU Best tree parallelization result. For training, by utilizing lockless update, we manage to improve the efficiency of CPU parallelization dramatically. In our results, GPU can speed up more than 62 times compare to the CPU Baseline. Considering the total training time, at 2-layer, 2 GPUs matches about a 32 thread computer; and at 3-layer, 2 GPUs is 2.5 times faster than a 32 thread computer.

In real-time matching scenarios, inferencing time is really important (can't let the opponent wait too long for one move). GPUs can achieve relatively fast inference time, and also fast training time compare to CPU alternatives, which makes it a great choice for parallelization this problem.

9. REFERENCES

[1] M. Szubert and W. Jaśkowski, "Temporal difference learning of n-tuple networks for the game 2048", 2014 IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8, August 2014.

[2] E. Melkó and B. Nagy, "Optimal strategy in games with chance nodes," Acta Cybernetica, vol. 18, no. 2, pp. 171–192, January 2007.

[3] StackOverflow. What is the optimal algorithm for the game, 2048? [Online]. Available: http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22674149#22674149

[4] Wu, I-Chen & Yeh, Kun-Hao & Liang, Chao-Chin & Chang, Chia-Chuan & Chiang, Han. (2014). Multi-Stage Temporal Difference Learning for 2048. 366-378. 10.1007/978-3-319-13987-6_34.

[5] J. Veness, "Expectimax Enhancements for Stochastic Game Players," B.S. thesis, School of Computer Science and Engineering, University of New South Wales, 2006.

[6] Rigie G. Lamanosa, Kheiffer C. Lim, Ivan Dominic T. Manarang, Ria A. Sagum, and Maria-Eriela G. Vitug "Expectimax Enhancement through Parallel Search for Non-Deterministic Games", International Journal of Future Computer and Communication", vol. 2, no. 5, October 2013

[7] Tsan-sheng Hsu. Parallel Game Tree Search [Online]. Available: http://www.iis.sinica.edu.tw/~tshsu/tcg/2013/slides/slide11.pdf

[8] Kamil Rocki and Reiji Suda, "Parallel Minimax Tree Searching on GPU" PPAM 2009, Part I, LNCS 6067, pp. 449–456, 2010

[9] Kamil Rocki, Reiji Suda, "Large-Scale Parallel Monte Carlo Tree Search on GPU" IEEE International Parallel & Distributed Processing Symposium, 2011

[10] Daniel Shawul, Experiment on gpu tree search using the game of hex [Online]. Available: https://github.com/dshawul/GpuHex