2048 Game Solver on GPU

Prashanth Soundarapandian
111498721

Rakshith Reddy Polireddy
111498989

Abstract

2048 is a stochastic single player game, originally written in java script for playing in a web browser, but now largely played in mobile games. This report discusses the applicability of ExpectiMax search to solve the problem using a serial implementation and a parallelized version that runs using GPU threads. We will demonstrate the potential speedups obtained with using GPU and present the results with varying heuristics that determine the score of the terminal state. our AI agent was able to achieve a highest tile of 8192.

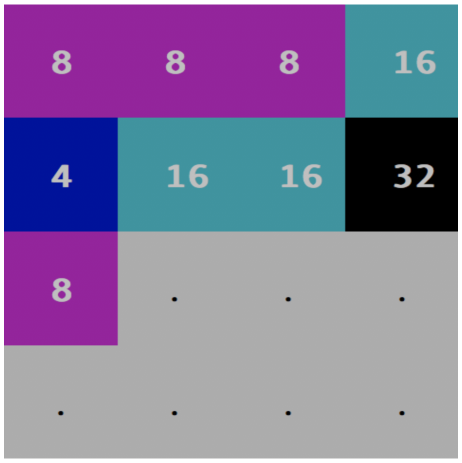

Figure 1: A 2048 Game State with four permitted operations (Left, Right, Up, Down)

Introduction

2048 is a sliding game played on a 4 X 4 grid which is similar in style to rush hour and 15 puzzle game. It is not a puzzle, as you cannot calculate the optimal path to the goal state. The stochastic nature of the game calls for a strategy to maximize the score.

Motivation

Developing AI agents for games has been a central topic in Artificial Intelligence and most of the agents uses the search through the game state space with some incorporated logic. Nevertheless, the state space for some games is huge and it would take a lot of time to search using a single core machines. To develop an AI agent for 2048 game would require a search through the game state and the branching factor for the player is 4 (up, down, left, right) and the branching factor for the computer is dependent on the number of the free tiles as the computer can place a tile at random in each of the free tiles. Therefore, it is clear the state space is going to be huge and is practically in-feasible to explore the complete game states. This calls for machines with superior resources and parallization to compute the next step faster. This 2048 puzzle is proven to be NP-complete[2].

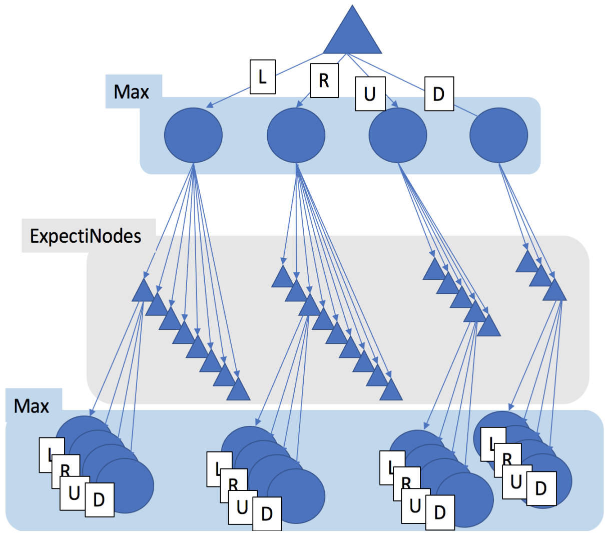

Figure 2: A 2048 Game State branching factor with ExpectiMax Search with four permitted operations (Left, Right, Up, Down) at Max node and no of free tiles at the expectnode.

Previous work

Mini-Max

The first AI for 2048 appeared in the year 2014[1], where the game was treated as a adversarial; the AI performs depth-limited Mini-Max searching, assuming the opponent will place the random tile as bad as possible. While this approach is reasonable; plan for the worst, the AI is making a decision on a even that is highly unlikely to occur, which results in a sub optimal solution. This work considers an evaluation function that is based on the *monotonicity* of the terminal state at the max depth. *Monotonicy* describes that the tiles increase or decrease in a uniform manner, with adjacent tile being close powers of 2.

Expectimax

Robert Xiao implemented an AI agent that used the $Expectimax$ search to play the game. It is similar to the Mini-Max Algorithm that is similar to a depth-limited search algorithm and uses a similar heuristic. Unlike Mini-Max search, where we find the maximum of the minimum at each level, in $Expectimax$ we find the maximum of the average.The Expected score of state is the average of the probability of that state occurring in the subsequent step.

Goals

- *Our goal is to develop an AI agent on GPU for 2048 game using Depth-Limited Expecti-max algorithm.*

  Explanation/Justification: Generally, Min-Max algorithm with alpha-beta pruning is used for two-player zero sum games, but 2048 is a stochastic game in which randomization is involved in computer placing a tile at any one of the free tile positions. Expecti-max algorithm is used when randomization is involved to account for all the possibilities. Therefore, it turns out that expecti-max is the algorithm that we need to use to search through state space. Also, since the whole state space cannot be explored, we use depth limited expecti-max algorithm. Exploring only until a depth does not affect the agent adversely as the probability to reach a particular state reduces as we go down the tree. However, it is difficult to use alpha-beta pruning in Expecti-max as it is complicated. We can also use Min-Max algorithm with the assumption that the computer plays in a way that it tries its best to reduce the score of the max-player. Since, the state space is huge, the complete state space cannot be explored. Also, if we use depth-limited Min-Max algorithm it would not allow the agent to perform good in all the cases. These are the two reasons which proved that using Depth-limited Expecti-max is better than the others mentioned.

- *To analyze, how the serial and parallel players perform when time bounded.*

  Explanation: We vary the depth so that, we could see the agent making atleast 5 moves per second. Time take to explore a given depth is not constant as the branching factor varies continuously. Since parallel player explores the paths parallelly, it explores more depth in the same time compared with the serial player and thus we analyze how the serial and parallel players perform in terms of the highest tile created by the players, time taken to reach the goal and other results. Also, the depth explored by the parallel player should

- *To analyze the times taken, parallelism achieved and other results to solve the 2048 game by the serial player and the parallel player when depth bounded.*

  Explanation: We could fix the same depth for both the serial and parallel players and compare the times taken by both the players to reach the goal. This will provide us the time efficiency achieved, parallelism and other such results.

- *To analyze all the above variants for all the methods, i.e perform the above analysis on minmax algorithm with alpha-beta pruning and memory bounded IDA star algorithm.*

  Explanation: We perform both the time bounded and depth bounded analysis with serial and parallel players that we performed above for both the minmax algorithm with alpha-beta pruning and memory bounded IDA star algorithm.

Modified Goals

As discussed in the second section (Original Goals), since 2048 involves randomization, the agent may perform better using expecti-max algorithm, therefore we wanted to try implementing it using Depth-Limited Expectimax as well as other algorithms stated in the original goals and compare them.

Our Strategies

We have built two versions of the game that is based on parallelizing the $Expectimax$ search strategy and optimizing the code to run on GPU. We study the significance of the varying heuristics that is applied to the terminal state and present the maximum score each AI agent is able to achieve.

We also extended Robert Xiao's work on serial $expectimax$ search by evaluating the efficiency of the parallel $Expectimax$ search on the Comet supercomputer and present the results.

Averaged Depth-Limited Search

Average Depth Limited Search works by running multiple simulations of the $Expectimax$ search strategy. This strategy maximizes the likely outcome with each run by not taking into account all the probability values present in the original $Expectimax$ search process, instead only those which are more likely to occur.

Evaluation Functions

Evaluation functions are applied to the terminal state; the state of the 4X4 board at max-depth permitted by the memory restrictions. This forms the crucial part of the $expectimax$ search algorithm. The max tile formed has a huge dependence on the heuristic function chosen.

Monotonicity

This heuristic tries to ensure that the values of the tiles are in the increasing or decreasing order in the top/bottom and left/right rows/column. This heuristic ensures that the highest valued heuristic is clustered in the corners. It will prevent the smaller valued tiles from getting orphaned and will keep the board very organized.

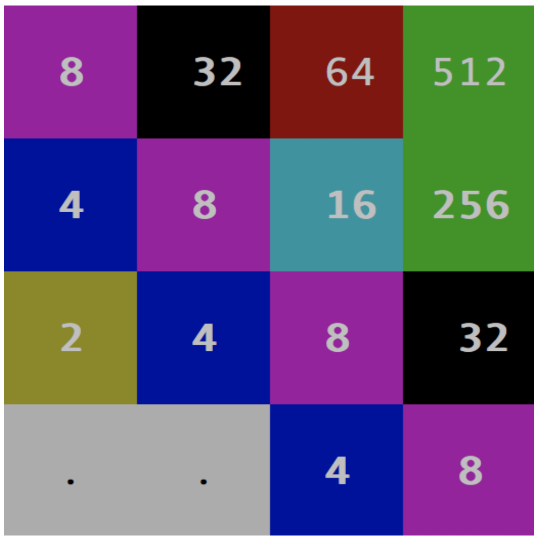

Figure 3: A 2048 Game State with Monotonicity heuristic

Smoothness

This heuristic aims to create structures in which adjacent tiles are decreasing in value and tries to ensure that the neighbouring tiles are as close in value as possible.

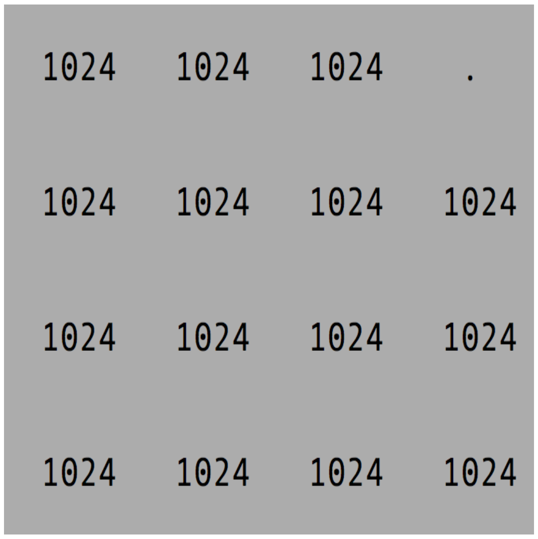

Figure 4: A 2048 Game State with Smoothness heuristic

Free Tiles

The third heuristic penalizes for having too few tiles in the terminal state at the maximum depth.

Searching through the search space by incorporating these heuristic yields interesting and unexpected results.

Pseudo Code - Serial Version

The pseudo code for the serial version is as follows :

```
Result: The next move from the set { left , right, up,
        down }
Expectimax
  (matrices,level,type)://type(max:1,expecti:2)
if level == max_level then
    for matrix in matrices:
        matrix.heuristic = calculate_heuristic(matrix);
else
    if type == 1 then
        create 4 children ( left , right, up ,down );
        Expectimax (4 childs,level+1,2);
        calculate max of four directions;
    else
        Create N copies of each matrix, where N is the
          number of vacant spots;
        //This is the branching factor at this level;
        Expectimax (M matrices,level+1,1);
        Take average weighted sum of all probabilities;
    end
end
```

Algorithm 1: Serial Expectimax Stratergy for 2048 Game.

Serial Version of the game is a straight forward implementation of the expectimax search. There are two agents which operate at alternating level. The first agent is the max node agent which picks the maximum node among the four children which are generated. The next agent is the expecti agent which operates at the subsequent level picks each vacant spot to fill the tile with some probability. The branching factor at the expecti agent is thus the total number of free tiles available from the previous level. In this implementation we have assumed that the probability of picking a free tile at a position is $\frac{1}{n}$; where n is the number of free tiles in the given matrix. Also this is repeated across all the matrices thereby summing the total number of children generated at the subsequent levels.

One of the disadvantage of using the expecti-max strategy is that due to large branching factor, memory usage is exponential. Unlike Mini-Max strategy there is no possibility of alpha-beta pruning because we cannot effectively determine which nodes to purge unless we look at the node at least once.The probability of the node getting picked in the next layer of max node step is dependant on the edge weight of the expecti-node. One way this problem could be addressed is by storing in the matrix the power of 2 value ; the value to be displayed as the tile value is $pow(2, power\_value)$. The next step of optimization is use 4bits to encode each power value, as we can represent upto 2^{16} values with 4 bits. This brings the total size of each matrix to just 4bytes; 1byte for rach row, thereby saving a lot of memory. Potential speed up is also achieved because of using less memory as the iteratoion to convert the matrix into its children is faster.

Pseudo Code - Parallel Version

The pseudo code for the Parallel version is as follows :

```
Result: The next move from the set { left , right, up, down }
Expectimax
  (matrices,level,type)://type(max:1,expecti:2)
if level == max_level then
    len1 = len(matrices)
     calculate_heuristic$<<< len, 1 >>> (matrices)$;
    //The above line launches a new kernel with "len1" blocks such that the value of the matrices is calculated at once in parallel
else
    if type == 1 then
        $create\_copies <<< n, 16 >>> (matrices, child\_matrices)$
        //this will copy all the 16 elements of a matrix into the child_matrix at once in parallel, this is performed 4 times.
        cudasynchronize();
        $moveleft <<< n, 4 >>> (child\_matrices)$;
        // here 4 indicates that each thread operated on a row $moveright <<< n, 4 >>> (child\_matrices)$;
        $movedown <<< n, 4 >>> (child\_matrices)$;
        // here 4 threads indicates that each thread operated on a row.
        $moveup <<< n, 4 >>> (child\_matrices)$;
        $cudadevicesynchronize()$;

        //this will launch all the kernels and wait for them to finish, i.e, all the right left up and down are performed in parallel with each thread assigned to a row.
        Expectimax$<<< 1, 1 >> (4childs, level + 1, 2)$;
        calculate max of four directions;
    else
        Create N copies of each matrix, where N is the number of vacant spots;
        //This is the branching factor at this level;
        Expectimax$<<< 1, 1 >>> (matrices, level + 1, 1)$;
        Take average weighted sum of all probabilities;
    end
end
```
Algorithm 2: Serial Expectimax Stratergy for 2048 Game in GPU.

Some of the challenges faced while implementing the parallel version of the game using expectimax search stratergy was to:

- Understand CUDA and how it operates.
  To effectively use the GPU's computing power we need to allocate memory in the GPU.
- Understanding of what zero-sum games are and the methods to design agents for them using minmax, expectimax etc and verifying if parallelism is possible or not.

The parallel implementation of the serial implementation is tricky given the fact that recursion is not supported by default in GPU and requires dynamic parallelism modules be imported for recursive operations to operate with ease.

The serial version of the expecti-max search is parallelized using building GPU kernels with threads. At the first level when the expectiMax tree is about to explode the branching factor is 4 as this level is the max level where four children are generated. At this level a serial version would 4*16 operations to create 4 children as there 16 elements in each of the matrix and we have to create 4 copies. In the parallel version, use 4 threads to create 4 copies in parallel. So we need 16 operations to compute 4 children. This can further be parallelized by using 16 threads for creating 4 copies where 4 threads operate on each of the matrix. This is possible because of the property of the 2048 game, where each operation of left,right, up, down is a dependant only one row/column. So when 4 threads operate in parallel we only take one cycle to perform the copy/move operation. Since the 16 threads operate on 4 threads in parallel, we see that the entire copy operation occurs in just n cycles; where n in this case is 4.

At the expecti-level we were not able to acheive such a degree of parallelization because the branching factor in this case is not uniform as the number of children that gets generated at each level depends on the state of the 4X4 board. So achieving such a degree of parallelism not possible because of the non uniform branching factor as described above. So that segment of the code runs serially because of warp divergence.

Also, we observed that an anomaly with the $cuda\_malloc$ that the time taken to allocate memory was significantly more than the time taken to allocate memory in CPU.

GPU Memory Model

GPU has a much more restricted memory model where free memory is computed just before computation and the memory is limited during computation. In this 2048 implementation we use the global memory which has the following properties:

- Read-only during computation
- Write-only at end of computation (pre-computed address)
- Read/write in GPU world only

One disadvantage of using GPU memory model is that the virtual memory in GPU does not exist and therefore the

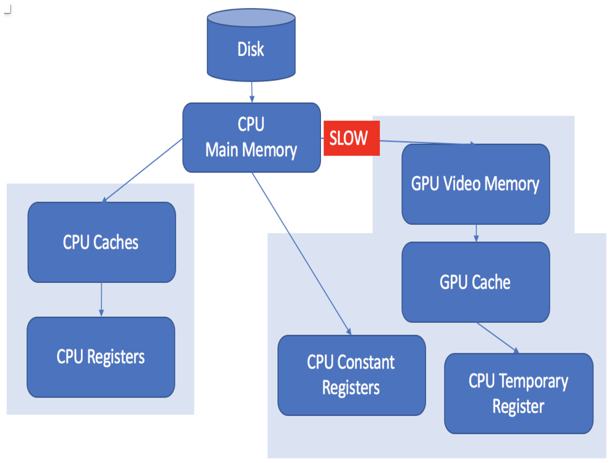

Figure 5: CPU and GPU memory segments overview.

memory incentives like the copy on write mechanism does not exist. So the amount of memory that gets allocated is diectly on the RAM which access tends to be very slow tends to be slow.

Results

CPU vs GPU

Fig 6 describes the average time taken at depths 2 to 7 over 100 games to make a single move. This graph shows that at lower depth of the expectimax search tree, we see that the CPU serial implementation performs faster and the GPU suffers because of the its memory model. However, we see that time taken to GPU to make a single move grows faster as we seek more memory that is more aligned and reusable . We conclude that at Higher levels the GPU code outperforms the CPU version by large value.

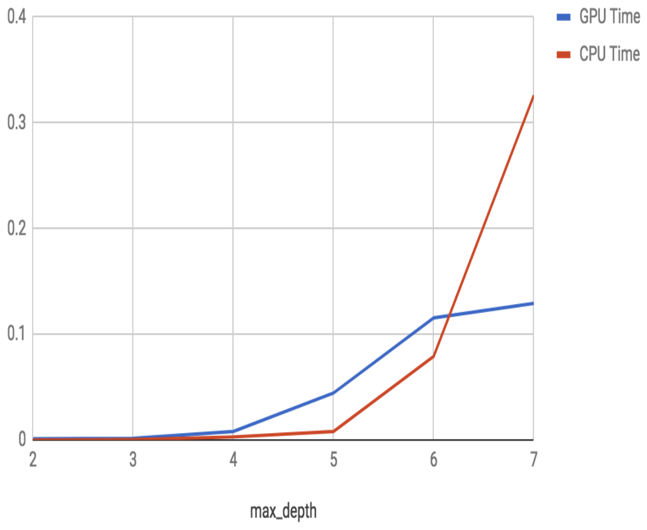

Figure 6: CPU vs GPU time (seconds) taken to make a single move

Max Score Tile

Our AI agent was able to achieve a max score of 4096 once.

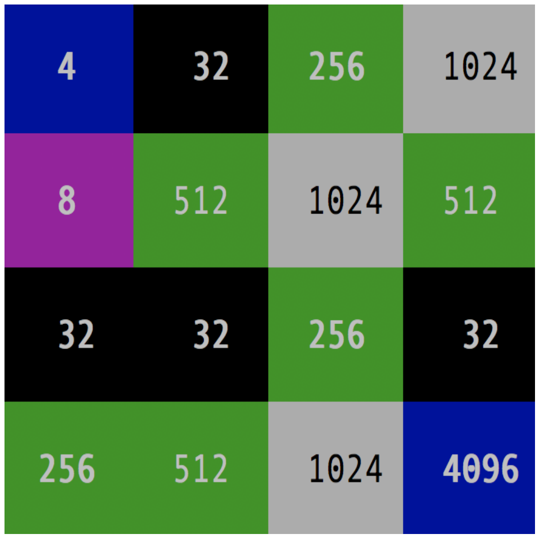

Figure 7: max tile reached

Our AI Agent was also able to reach 1024 tile 90% of the time and the 2048 tile 80% of the times. This result was calculated over 100 games played by the agent both on CPU and GPU. The max tiled reached was independent of whether the algorithm was run on GPU or CPU.

Max Score

One more metric that evaluates the usefulness of our implementation is the scoring component that is estimated as follows. When a tile merges with another tile the score is incremented by the sum of these tiles. For ex. When a tile 4 merges with another tile 4. The score gets incremented by 8. The following graph shows the score obtained by the AI agent when the max depth was kept from 2-7 and the performance was seen to the the most optimal at depth 6.

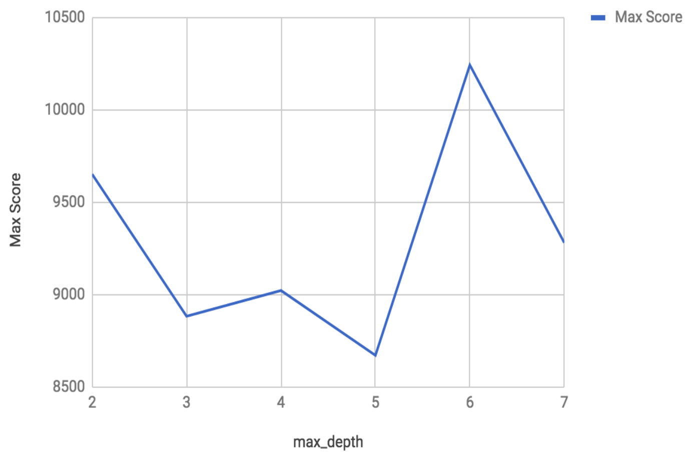

Figure 8: max score reached at level 6

Future Work

Evaluation Functions

We have conceived many heuristic functions that may get better results.The most promising has been based on 1 dimensional monotonicity whereby we reward boards that are monotonic in an 'S' shape, rather than each row and column being monotonic. This seems to be the strategy favoured by human players. We have implemented this this evaluation function, and it does generate the correct board patterns. But his can be improved to discover better heuristics that could still solve the game faster. Petr Morvek [4] version of the expectimax search based 2048 game solver used heuristic weights using a "meta-optimization" strategy (using an algorithm called CMA-ES), where the weights themselves were adjusted to obtain the highest possible average score.

The effect of these changes are extremely significant. The algorithm went from achieving the 16384 tile around 13% of the time to achieving it over 90% of the time, and the algorithm began to achieve 32768 over $\frac{1/3}{}$ of the time (whereas the old heuristics never once produced a 32768 tile)

Memory Trace

The algorithm can be further improved to consume less memory using bit manipulation and thereby reducing the memory pressure on the GPU, which we believe cause a significant impact on the overall efficient of the agent.

Alternative Strategies

[5]PhiPhilip Rodgers and John Levine paper talks about alternative strategy that could possible solve the game faster and leave a better memory trace on GPU.

Monte-Carlo Tree-Search

MCTS has three key properties that make it interesting.

- It can run for virtually any amount of time and still return a result; the longer it runs, the better the result.
- It does not require an evaluation function, as the roll-outs do this job. The random nature of the roll-outs ensure no moves are ruled out.
- It produces asymmetric trees, effectively pruning poor paths allowing for deeper searching of the paths with greater potential. It is the asymmetry of the trees produced that allows MCTS to concentrate its effort on the potentially good moves, but we found that the trees produced whilst playing 2048 were symmetrical.

Szubert and Jaskowski[3] successfully used TD learning together with n-tuple networks for playing the game 2048. In their paper, they first improve their result by modifying the n-tuple networks. However, they observe a phenomenon that the programs based on TD learning still hardly reach large tiles, such as 32768-tiles (the tiles with value 32768). In their paper, they propose a new learning method, named multi-stage TD learning, to effectively improve the performance, especially for maximum scores and the reaching ratio of 32768-tiles. After incorporating shallow expectimax search, our 2048 program can reach 32768-tiles with probability 10.9%, and obtain the maximum score 605752 and the averaged score 328946. Their program outperforms all the known 2048 programs up to date, except for the program developed by the programmers, nicknamed nneonneo and xificurk, which heavily relies on deep search heuristics tuned manually. The program can reach 32768-tiles with probability 32%, but their implementation runs about 100 times faster.

References

[1] Mini-max AI. URL: http://ovolve.github.io/2048-AI/.

[2] Ahmed Abdelkader Aditya Acharya Philip Dasler. URL: http://cs.umd.edu/~akader/files/CGYRF15_2048.pdf.

[3] Wojciech laskowski Marcin Szubert. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6932907.

[4] Petr Moravek. *Nneonneo*. URL: https://github.com/xificurk/2048-ai.

[5] Philip Rodgers **and** John Levine. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6932920.