CUDA Parallel Programming Project

Minimax Game Tree Search

Caleb Klaus, Ioan Iordanov (UIUC)

Background

We implemented a game which works as follows: 6x6 game board where each space has a value. Two teams, Green and Blue; blue goes first and it is turn based. Your score is the total of all the values of the spaces you occupy. Team with highest score at the end wins. We created 5 different maps and ran our AI vs AI algorithm on all 5 when collecting results. There are 2 possible options when making a move.
First Scenario: Take an unoccupied space on the board and none of your vertical or horizontal neighbors are yours. This move ends your turn.

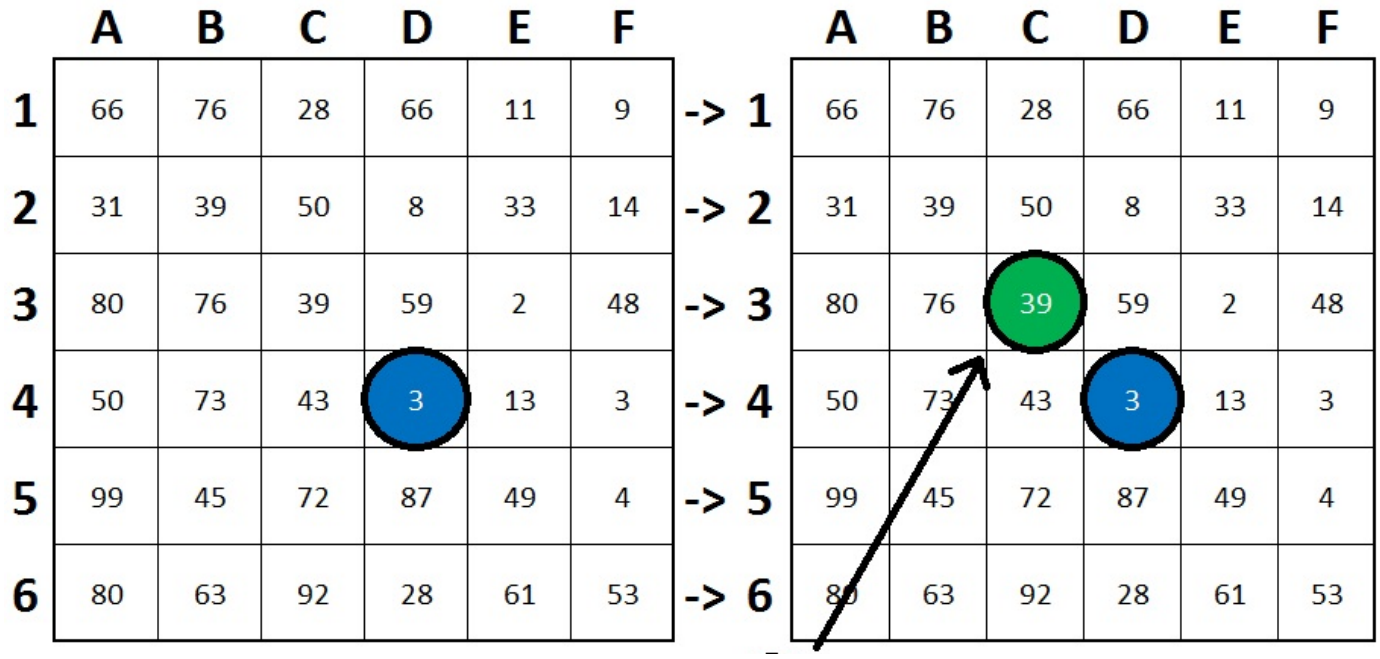


Illustration from CS 440 website

Green take a turn and adds 39 to their total score. Blue is unaffected.

Second scenario: Take an unoccupied space on the board and at least 1 of your horizontal or vertical neighbors is yours. If this condition holds then all vertical and horizontal neighbors now belong to you.

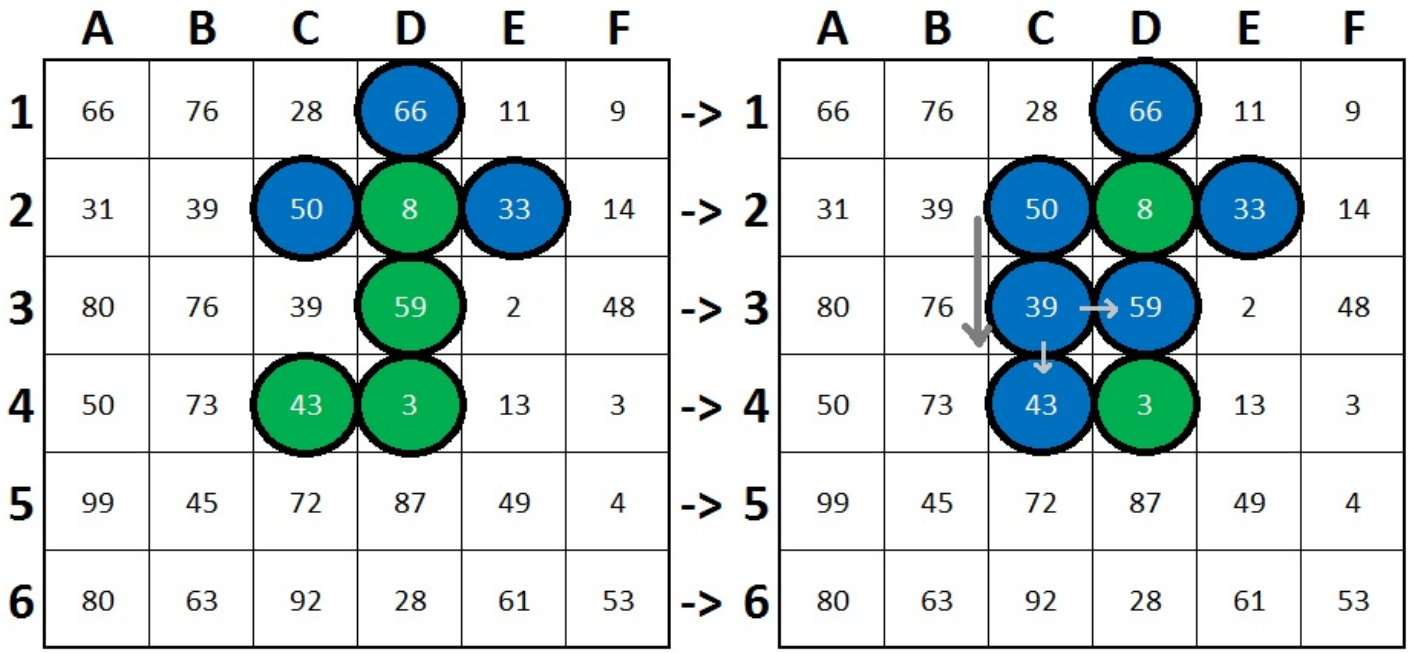


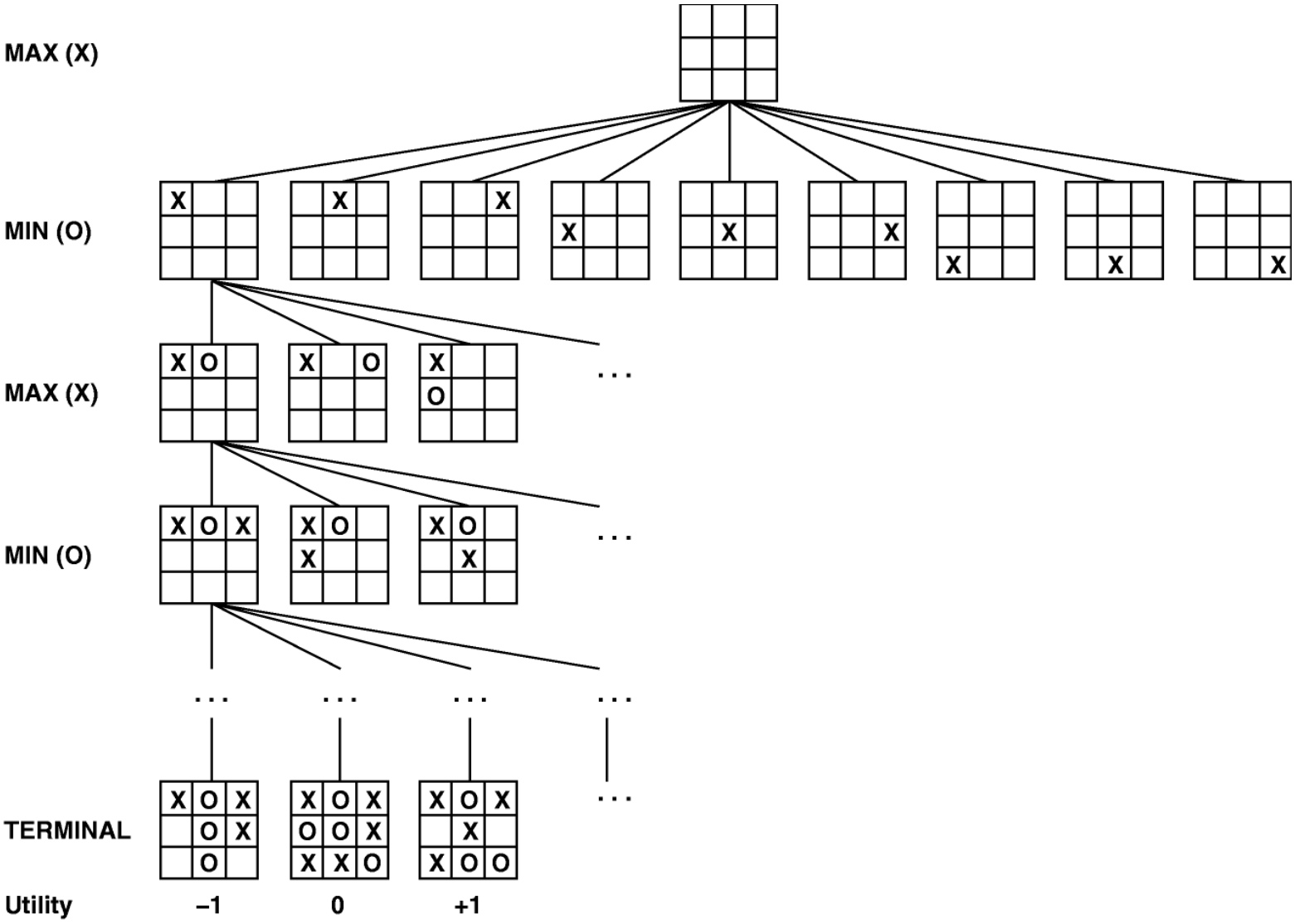
Illustration from CS 440 website

Blue makes a move and adds 39+59+43 to their total score. Green loses 59+43 from their total score.

The AI - Minimax Search

For the AI, we used a simple minimax algorithm to determine where to go next. The minimax strategy essentially minimizes the possible loss for the worst case scenario, and is optimal against an optimal opponent (an opponent using the same strategy). Below is a simplified example with tic tac toe which is a 3x3 board. It is just a data tree that grows exponentially

with each level and represents all possible moves. Once the algorithm gets to a terminal state or reaches its depth limit, it returns a value, and this gets propagated up the tree, resulting in the best move then being returned to the root.



The AI is better the deeper the tree is, however the cost of going deeper greatly increases with each level. For our game, the evaluation function used at the depth limit is found by computing the total scores of each team at that time and finding the difference. Max team will try to maximize this value, and min team will try to minimize this value.

Motivation

For a 6x6 game board, there will be a maximum number of 36*35*34 = 42,840 leaves in the game tree for depth 3. The CPU could handle this in a reasonable amount of time. However, going just 1 level deeper creates over 1.4 million leaves, which is too much for the CPU to handle in a reasonable amount of time. Since what happens on each game board is localized (each turn amounts in at most 5 game locations handled), it would not be very efficient to have multiple threads working on 1 game board at a time. Instead, we decided to run the CPU for depth 3 which generates a good amount of leaves, and stored these leaves. Next we copied the leaves to the GPU and had each thread work on a leaf and expand it. The results were copied back to the CPU, which then used those values to propagate up the tree.

Our Goals

- Improve performance time of depth 4 AI, by using CPU for depth 3 and GPU for the additional depth 1 for a total depth of 4.
- After only 8 moves are left, use CPU to go all the way to the end since there aren't enough leaves to justify the use of the GPU
- Optimize thread distribution for particular hardware
- Optimize memory accessing and thread coalescing

Memory Coalescing

First we organized the CPU generated leaf storing array as the first 36 elements being the first game state, the next 36 elements being the next game state… etc. Since all threads in a warp all try to access the first element in their game state during the first iteration of the loop, it would take 32 memory accesses which are each 36 memory addresses away from each other. However, if we put all the first elements of each game state in adjacent memory space, each thread will be reading in a memory address which is adjacent to the memory address of another thread in the warp. This results in coalesced memory access and decreases the amount of memory address computations that need to performed, thus lowering the memory bandwidth and increasing performance.

Here is an illustration of a simplified version for a 3x3 game board with 6 game boards. Same colors represent memory accesses that will happen at the same time within a warp.

All game states stored consecutively (no coalescing)

1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2

All same game board locations stored consecutively

1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	4

Testing Hardware: Nvidia GTX 980 (Maxwell GM204)

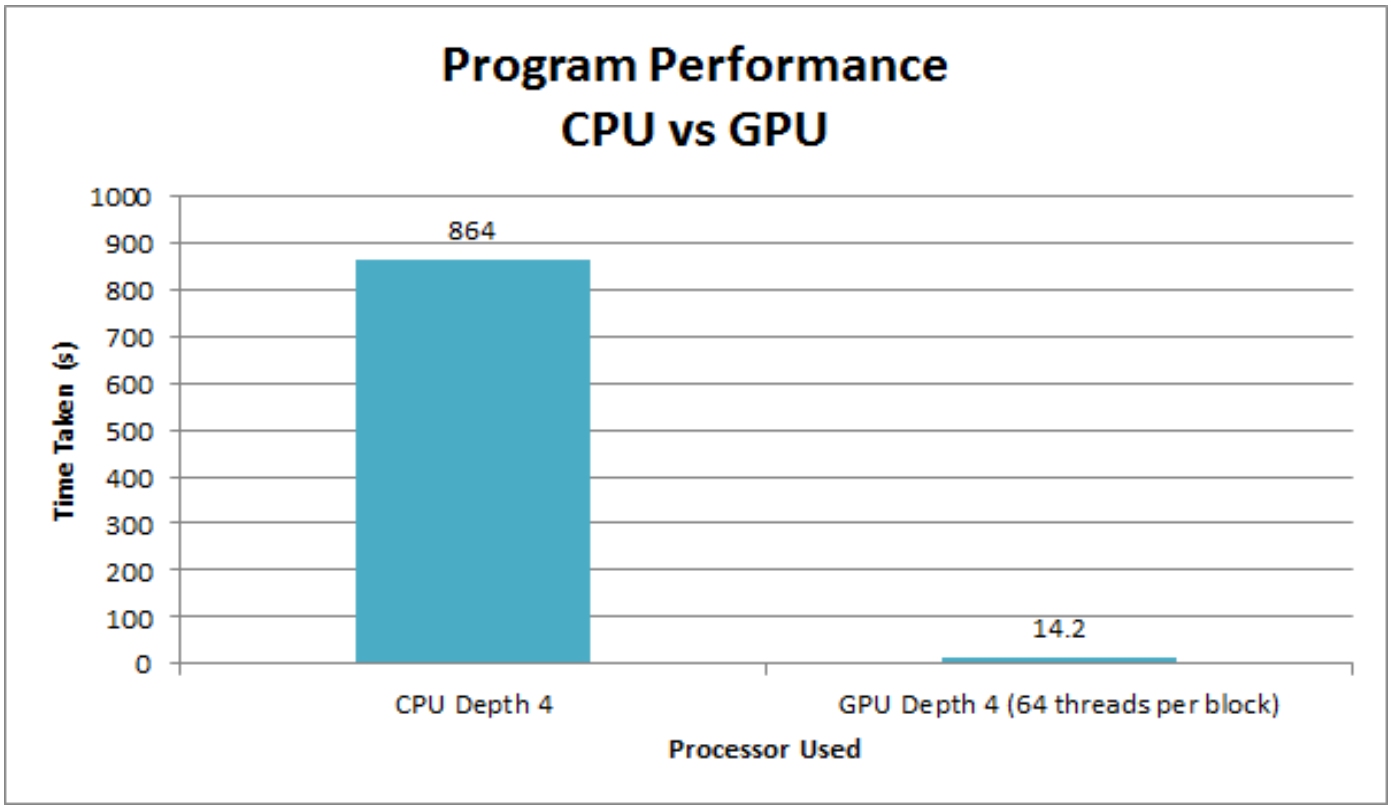
Below are the hardware specs of the GPU we used for testing. This is important to note as testing results will be different on other GPUs.

GPU	GeForce GTX 980 (Maxwell GM204)		
CUDA Cores	2048		
Base Clock	1126 MHz		
GPU Boost Clock	1216 MHz	Memory	4096MB
GFLOPs	4612	Memory Clock	7010 MHz
Compute Capability	5.2	Memory Bandwidth	224.3 GB/sec
SMs	16	ROPs	64
Shared Memory / SM	96KB	L2 Cache Size	2048KB
Register File Size / SM	256KB	TDP	165 Watts
Active Blocks / SM	32	Transistors	5.2 billion
Texture Units	128	Die Size	398 mm²
Texel fill-rate	144.1 Gigatexels/s	Manufacturing Process	28 nm

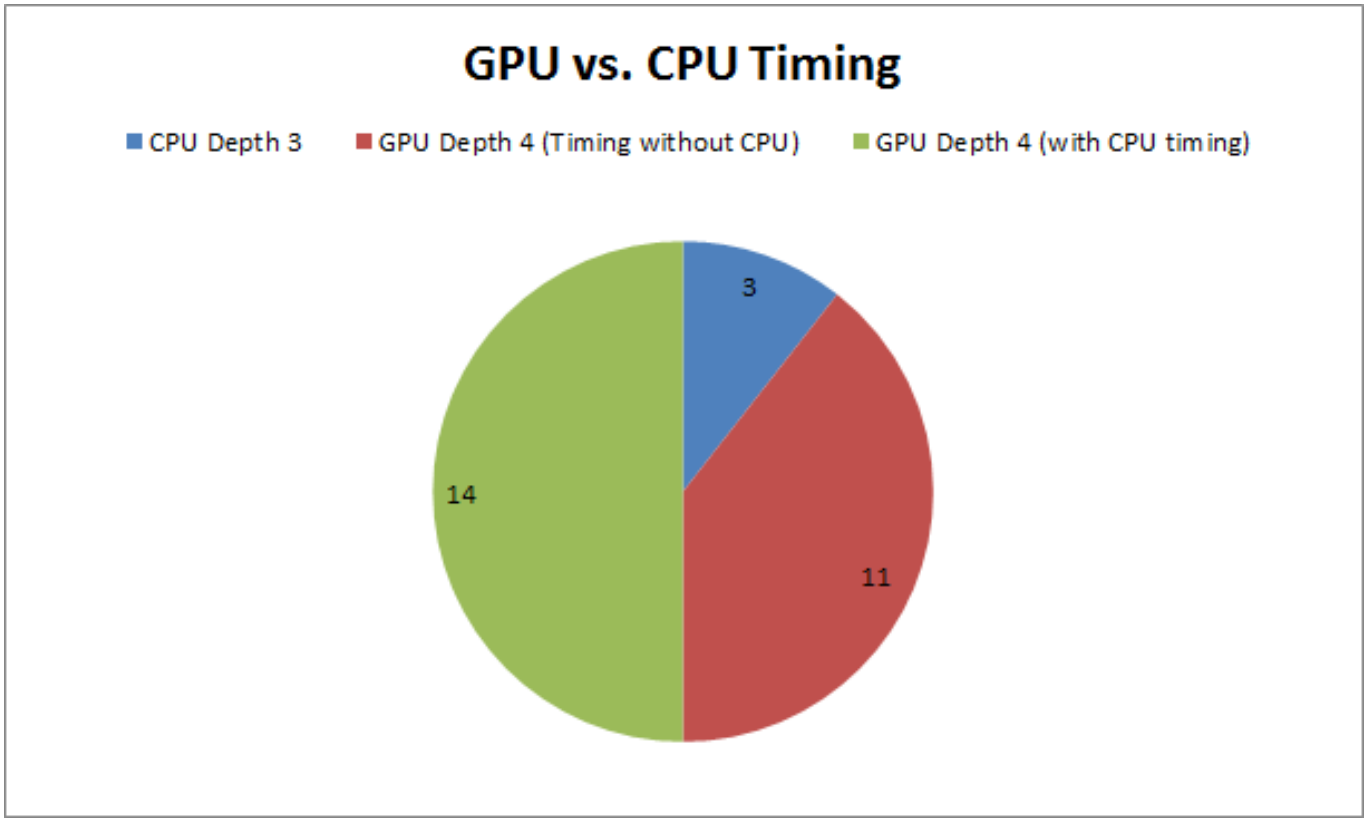
- There are 16 SMs per GPU.
- In every SM, there are 32 active thread blocks
- And for every SM there are a maximum of 2048 threads
- In order to fill up all the SMs to maximum capacity there needs to be 2048 / 32 = 64 threads per block in order to use all 32 blocks per SM.

Results

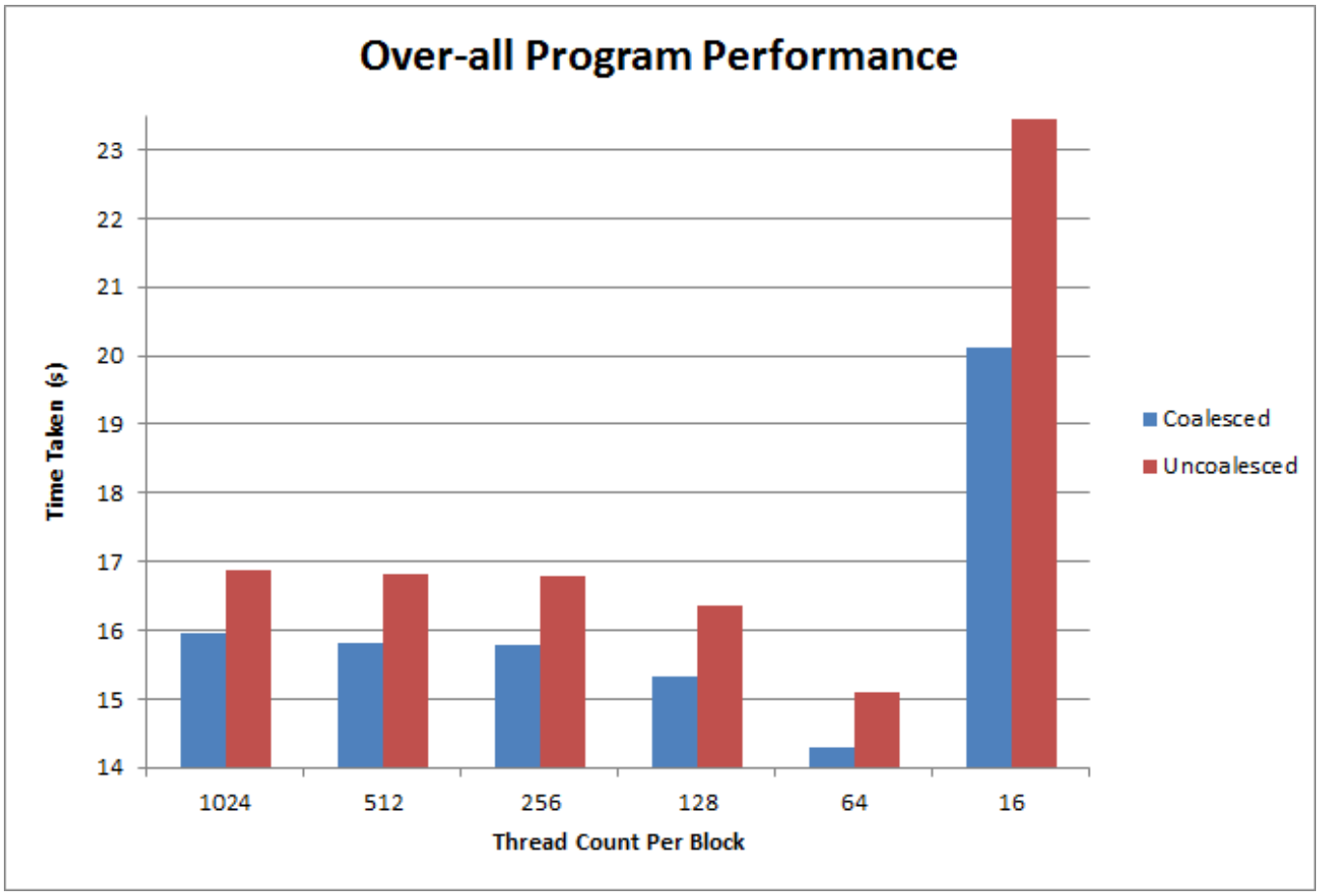
First we collected results for CPU vs GPU performance. The GPU resulted in roughly 60x speedup.



The GPU by itself took 11s, vs 3s for the depth 3 CPU part



Next we collected results for differences in using coalesced memory vs not using coalesced memory, and for difference in the number of threads used in a block



The results are based on the average after executing each case several times. The time could fluctuate by a second or so depending on when the program was run. However, the general trend was that memory coalescing resulted in a ~5% speedup. Each thread in a block performs 36 global memory ready, and the maximum number of global memory reads for just 1 turn is 36*35*34*36, which is over 1.5 million. The number of global memory writes is 3 per thread in a block, resulting in a maximum of 36*35*34*3 = 128,520. The max total number of global memory access per turn is 1.67 million.

Further Possible Improvements

There are several things that could be tried to possibly improve performance.

- Minimize control divergence by optimizing the game logic. Our game logic was relatively simple compared to more advanced games, so this may not result in a huge gain in performance time.

- Increase the depth for the GPU. Instead of handling just 1 game tree level on the GPU, perform 2 or 3 levels on the GPU. The biggest limiting factor here is memory available on the GPU, which is why we did just 1 level. However, it may be feasible to do a deeper search once the board starts to fill up, as the game tree will not grow as fast.

- Have the CPU do some computations while the GPU is doing its work. Right now we call cudaDeviceSynchronize after the kernel launch to ensure all the data computed in the kernel is ready for the CPU to use. However, we could have the CPU be expanding its own tree while the GPU executes which would be independent of what happens on the GPU.

Source code for this project can be found at the following link:
https://github.com/keikurausu/CS-483-final-project/blob/master/kernel.cu