A parallel design of computer Chess engine on CUDA-enabled GPU

Rohan Walia

Background:

My project is centered around speeding up a Chess searching algorithm by running the search primarily on the GPU. The problem with a Chess engine that only runs on a GPU is that you have a large depth first search running on a single core and the problem, traditionally, is exponentially harder as we increase depth. The reason that we have an exponential problem is that each new depth increases the number of nodes by roughly 35 times, depending on the position, as for each node of the previous depth we gain 35 new possible boards. My idea then was to launch hundreds of GPU cores to parallelize the problem and hopefully get better performance than CPU searches.

My motivation behind this project is that I've really gotten into learning about and playing Chess in the last few months. Chess has been an escape for me and my friends and I think anytime I can combine schoolwork and fun, I'll do a better job on the project. Another bit of motivation was watching a documentary about IBMs Deep Blue which used a series of chess specific integrated circuits to get the computing edge of Garry Kasparov, which made me want to build a chess engine that runs on some of our specialized hardware.

In terms of my relationship to the hardware, the Nvidia GPU was the most approachable non-standard piece of hardware that we could use and I didn't want to end up struggling for months and put out a weak project on something I barely understood. So, I ended up wanting to use the GPU as I thought it would be the right balance between learning and prior knowledge, as I knew the general ideas behind GPU programming, due to the research paper I presented (among others), but I would still have to learn a language, or at least half of a language. On another note, the Nvidia 3090 is the hottest computer part on the market and I thought only having Elijah fully utilize the GPU felt like a shame.

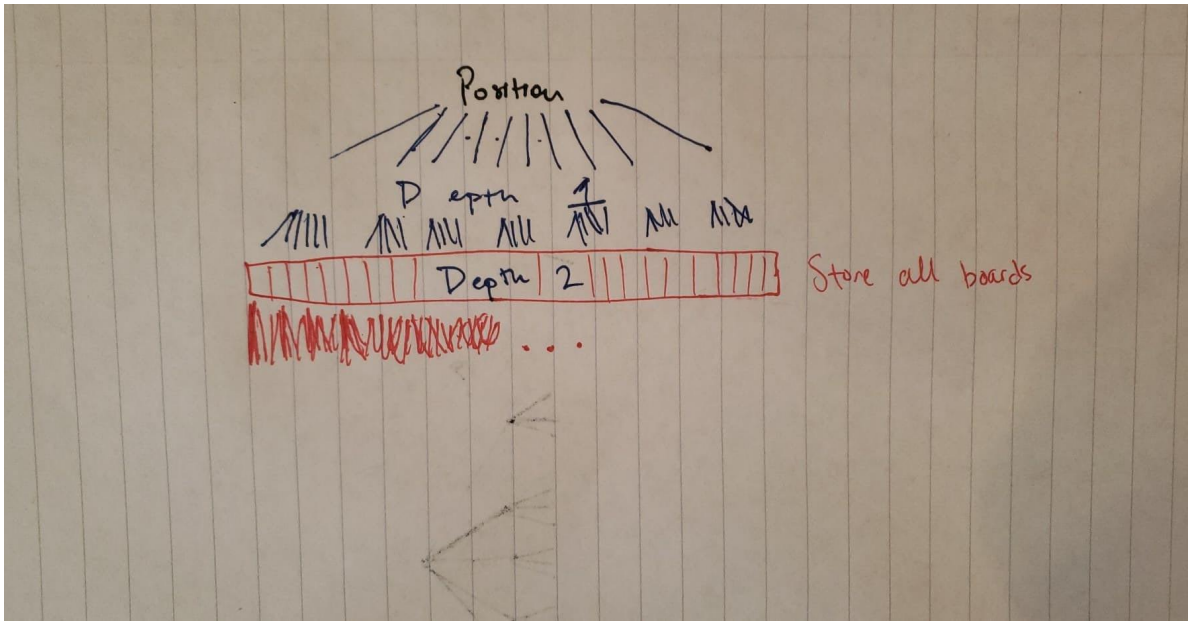
There was not any past research I could find that was using a GPU to run a Chess engine. The closest that I got was finding a Chess engine called Lella Chess Zero, built by a co-founder of the Stockfish main team, that was using a GPU to speed up some AI aspects of

the engine. I also found a paper that used a Monte Carlo Tree Search strategy on a GPU, to highly parallelize their Go search. While I was a little detoured by the lack of research, I thought there was enough evidence to continue forward and not have a complete failure of a project.

Proposed Solution (~2 pages)

First off we'll go over the most traditional Chess Engine. The most traditional Chess algorithm is a Depth-First MiniMax algorithm. A very basic implementation of this idea is the function RegNegaMaxSearch in my program. The function first creates all possible moves from the given board, then recursively calls itself, with one less depth, until the depth count becomes 0, at which point the position is evaluated. From that evaluated position, the recursion moves upward, running a MiniMax style search as we go up.

The way that my algorithm works is by first going down to either 1 or 2 deep down on the CPU and storing all possible boards. From there the algorithm calls the total number of saved board states, CUDA threads that each run the traditional Chess algorithm described above.



If we look at the beautiful diagram above, we see a demonstration of the algorithm where the CPU depth is 2 and the GPU depth is 1. The Black lines represent the CPU run, which then creates the red array of all possible positions of depth 2. Then a CUDA thread is called on each element of the board array. Once all of the CUDA threads have finished their analysis, there are two ways we get the best move: If the CPU depth is 1, then all we need to do is run through the array, which has the best moves from that position stored, and find the moves with the lowest

evaluation score, as those boards were analyzed from our opponent's POV. So, we want the move/board that is worst for our opponent. If our CPU depth is 2, we run a MiniMax algorithm on our board positions and retrieve the best move from that analysis. Then we simply print out the best move.

The major drawback behind this strategy is the highly variable number of threads that can be started due to the highly variant nature of chess boards. For example, if we have a chessboard with a king and 2 pawns on each side, we max have 20 positions from depth 1 and 20 more from depth 2 giving us 400 threads. Whereas the typical middle of game position has around 35 different moves giving us 1225 threads, if we went down to cpu depth 2. This drawback isn't as bad as it seems as the lower the number of moves, the less analysis needs to be done, so in tests the endgame positions manage to keep up. The other flaw in this system is that we cannot have a CPU depth past 2 as then the number of positions we store that we have will usually exceed the number of CUDA threads we have available to us (35^3 = 42875). This means that we will have to delay CUDA kernel calls* which will significantly slow performance.

The reasoning behind choosing a NegaMax algorithm over an Alpha-Beta algorithm, using AI, Quiescence searching, Monte Carlo Tree Searching, or a variety of other stronger algorithms, is that NegaMax has the lowest development time and I didn't believe it was in the spirit of the project to spend most of my time learning and working on a chess algorithm, but rather it was in the spirit to guarantee and work on getting any Chess engine running on the GPU, and then from there improving Chess aspects. On another note, the NegaMax algorithm has a fixed amount of space that it takes up, so it is easier to calculate the stack size than other algorithms. This is important as our Chess engine is a recursive function and CUDA cannot calculate the stack size on the function, so it must be set manually.

The C dependencies of the program are stdio, time, stdbool, math, and string. These fairly standard dependencies are used all over the place, but mostly for the "low level" chess functions such as evaluating a position, adding moves to a board, etc. In terms of CUDA, the program requires the Thrust host vector library for its usage in holding the boards that are eventually searched by the GPU, as we need a dynamically sized array to hold the positions, as Chess boards have a highly varying number of possible moves and therefore positions. Since the program is in CUDA, the program needs an Nvidia CUDA capable GPU, CUDA, Nvidia toolkit, and the CUDA compiler nvcc.

- Links to source code and libraries
- Libraries:
- Thrust:: https://docs.nvidia.com/cuda/thrust/index.html#abstract

- Nvidia Toolkit/nvcc: https://developer.nvidia.com/cuda-toolkit
- CUDA installation: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html
- C Libraries:
    - Stdio: https://www.tutorialspoint.com/c_standard_library/stdio_h.htm
    - Time: https://www.tutorialspoint.com/c_standard_library/time_h.htm
    - String: https://www.tutorialspoint.com/c_standard_library/string_h.htm
    - Stdbool: https://pubs.opengroup.org/onlinepubs/009696799/basedefs/stdbool.h.html
    - Math: https://www.tutorialspoint.com/c_standard_library/math_h.htm
- NegaMax Algorithm: https://www.chessprogramming.org/Negamax
- Sunfish Algorithm (Not mentioned but helped): https://github.com/thomasahle/sunfish/blob/master/sunfish.py

* Ran out of time, not implemented. Tried a few solutions, couldn't get anything to work.

Evaluation (~2 pages)

The computer that was being used has a Nvidia GeForce RTX 3090 GPU and an AMD Ryzen 9 3900 12-Core Processor, which is on x86_64 architecture and running Ubuntu 20.04. The workloads behind my testing was a repository of fens of middlegames and endgames. I have compiled 30 combined positions to test, 20 middle games and 10 endgames. I tested the following combinations of CPU and GPU depths: (1, 3), (1, 4), (2, 2), (2, 3), (4, 0), and (5, 0).

Reproducion instructions: Assuming you have the dependencies mentioned above and the required files, the following will compile the code:

nvcc -Xptxas='-suppress-stack-size-warning' -o Hammerhead Hammerhead.cu

*"-Xptxas='suppress-stack-size-warning'" gets rid of the nvcc warning about the stack size not being able to be determined.

This will create an executable called Hammerhead, which can be run as follows:

./Hammerhead [cpuDepth] [gpuDepth]

The cpuDepth can be either 1 or 2 and the gpuDepth can go up to around 10, but the computer will most likely crash before that analysis is finished. If nothing is entered, the cpuDepth defaults

to 2 and the gpuDepth defaults to 3. If the gpuDepth is set to 0, then the regular NegaMaxSearch will be run on the position with no GPU intervention.

Once the executable is started, the system will prompt you for a fen, which is a common string representation of a chessboard. If you enter “test”, then the computer will pull the 20 middle and 10 endgame fens from “test.txt” and output the times, in order, to “out_file.txt”. If you would like to get a few fens, the best way is to go to https://lichess.org/analysis , play a couple of moves around and then copy the fen that is right under the board. In fact, you could play against Hammerhead by playing a game and then on your opponent's turn running the fen on Hammerhead and playing that move (if you play against Hammerhead, you can turn off the stockfish evaluation on the right hand side).

Here are the summarized results from the tests:

Depth Form	(4, 0)	(1, 3)	(2, 2)		(5, 0)	(1, 4)	(2, 3)
Avg End (s):	0.223	0.216	0.043		0.341	2.770	0.545
Avg Mid (s):	0.571	7.933	0.323		19.468	397.322	14.665
Avg All (s):	0.455	5.360	0.229		13.093	265.805	9.958
Avg Diff (s):	n/a	-4.906	0.225		n/a	-252.712	3.134

https://docs.google.com/spreadsheets/d/1BO2P-PxvFO8d6km98HgjmvpRV2cuvdnSXZ7Y3HmePIE/edit?usp=sharing

The Avg Difference is the average of the CPU time - GPU time on every fen.

The data is split up into two parts, depths that sum up to 4 and depths that sum up to 5. The reason being that a depth 5 algorithm has to search more nodes than a depth 4 algorithm on the same node, so comparing their times is near meaningless in this case. If we look at the two cases where the CPU depth is 1, we are approximately 11.78 times slower on the average case than the CPU only implementation when the total depth is 4 and when the total depth is 5 we are approximately 20.30 times slower. While slower than I expected, this makes sense as the number of threads that we are launching is roughly 35 and at best in the low triple digits, which isn’t nearly enough to overpower the CPU. When we get to a CPU depth of 2, this is where we see promising results. When the total depth is 4, we have a positive average

difference of 0.225, which while I doubt is statistically significant, that would mean that (2, 2) is approximately 2 times faster than (4, 0). The difference increased to 3.134 when comparing (2, 3) to (5, 0) which could mean that (2, 3) is 1.315 times faster than (5, 0) on average. This is impressive considering that, in both cases, the endgame fens are even and pull down the average.

The comparison to other chess engine speeds online is not relevant to the overall goal. The reason is that our algorithm has no cleverness, pruning, or other neat tricks to save time, which means that Sunfish, a python engine, with decent pruning is roughly the same speed as our engine running on just the CPU, so with decent pruning we could crush what is an otherwise strong engine. A valid comparison is how our Chess engine would stack up to humans. If we look at “THE IMPACT OF SEARCH DEPTH ON CHESS PLAYING STRENGTH”[1] by Diogo R. Ferreira, if we were playing a chess game where we have roughly 10 seconds for each move, our engine would be rated around 1900 Chess elo or a US Chess Federation Class A player (2). This means that we would have a significant chance of winning against 99.81% of the United States population, which on the whole is above the world average chess skill.

Conclusion

Making a CUDA Chess engine has been a moderate success to reasonable certainty, we can say that our engine, while running at CPU depth 2, is faster than its CPU only counterpart. We can also say that the engine as a whole has been successful as it can dominate the vast majority of Chess players around the world. In terms of the experience of working on the project, this is the class that I’ve learned the most from. The experience of soloing a project, with help from the class, for two months straight was significantly less terrible than I thought it would be. In fact, doing this project has boosted my confidence in me as a programmer as now I know I can work through serious sticking points and complete large scale projects.

In terms of future work on the engine and the “wish list”, I have yet to implement infinitely scaling CPU depth and there is a whole bunch of pruning, algorithm changes, optimizations to improve the strength and speed of the engine. I think a solid set of goals would be to implement Alpha-Beta pruning, CPU depth above 2, and connecting the project up to a better user interface.

Special Shoutout to Robert Crovella of Nvidia for being the best at writing about CUDA issues.
P.S. There is a tradition of naming Chess engines after fishes which is why mine is named Hammerhead

[1] http://web.ist.utl.pt/diogo.ferreira/papers/ferreira13impact.pdf