GPU-Accelerated program to play Go

Zachary Clifford

May 12, 2009

1 Project overview

This project seeks to parallelize an algorithm for playing the game of Go using NVIDIA graphics cards. Go, like chess, is a board game that is easy to learn but very difficult to master. It has been the subject of some research to create computer programs to play it, but success has been very limited. Presently the best programs can perform at an intermediate to strong amateur level. For larger Go boards, there can be upwards of 100 legal moves for each player, making exhaustive analysis of future board positions difficult.

One method for approaching the problem involves using Monte Carlo methods. Rather than attempting to analyze a board position for strength in ways that would likely be sub optimal, Monte Carlo methods utilize large numbers of simulations of the game. To evaluate a move, the algorithm runs thousands of games to completion assuming that move was made. Each game is played more or less randomly by both players and a winner is declared. The strongest move is the one that has the highest proportion of games ending in victory for the player.

This project adapted the program GNU Go to use NVIDIA graphics cards for evaluating moves. GNU Go has a collection of different engines for making moves that are used to decide on a move. One of its modes is an experimental Monte Carlo engine that uses a similar algorithm to the one described here. This project will replace that module with one utilizing the NVIDIA graphics cards. The GPU-based algorithm will be compared to the original one in terms of relative strength and speed.

2 Go rules overview

It is important to have a basic overview of the game of Go as played by this project. To keep memory requirements down, this project will address the 9 by 9 area scoring game. There are two players, white and black. On a player's turn, he may place one of his stones on any of the unoccupied spaces on the 9 by 9 board or pass. The game ends when both players pass in succession indicating that neither one believes that further moves will improve his position.

The game revolves around territorial control, "liberties", and "groups". A group is a collection of stones of the same color all touching each other. Diagonal adjacencies do not count. A group can be any number of stones including a lone stone. A group has liberties equal to the number of open spaces adjacent to the group. Again, diagonal adjacencies do not count.

When a player places a stone on the board, the board must be examined for enemy captures. If this play has reduced any enemy groups to zero liberties, all stones in those groups are removed

from the board. After enemy stones are removed, any friendly groups still having no liberties are also removed. This is rare because usually a player will not place a stone leading to suicide.

At the end of the game, the territory on the board is computed. In the area scoring game, each player receives one point for every stone he has on the board. When it is apparent to both players that a certain group would be captured were the game to continue, those stones are removed from the board. With those dead groups removed, the players each receive a point for every open space that his stones completely surround. The player with the most points wins.

3 Description of implementation

NVIDIA's CUDA framework treats a graphics card as a hierarchy of computational elements. Individual lightweight threads are run in parallel. Threads are organized into blocks in a Single Instruction Multiple Thread architecture. The threads in a block must each execute the same instruction, though they may operate on different data. Multiple blocks can execute at the same time to achieve parallelism. A chunk of code that runs on the graphics card is called a kernel.

The device also has a hierarchy of memory that is explicitly controlled by the programmer. Device memory is plentiful and slow, though it can be read more quickly when it is read in contiguous blocks meeting certain criteria. Shared memory is specific to a block, but is much faster. Each thread can have multiple registers that are even faster.

The basic Go rules have been implemented in NVIDIA's CUDA framework as a kernel. The code can take a board position and an active player as input. It will play the game to completion and declare a winner. It will not resign until it sees no more "good" moves for either side. It has some basic rules for what constitutes a "good" move so that the moves are not totally random. "Good" moves either have liberties of their own, will capture enemy pieces or will avoid endangering a friendly group by reducing them to less than 2 liberties.

The main kernel is broken down such that one CUDA thread is responsible for each of the 81 spaces on a Go board. A few more threads were added to improve alignment. The CUDA architecture almost requires the number of threads to be a multiple of its internal warp size for good performance. Especially on older cards, the performance suffers is the thread total is not a multiple of 32. This thread idea maximizes parallelism on each game and seems to be standard practice for CUDA matrix problems.

The board is represented as a matrix with 16 columns and 13 rows. This was done to align the matrix with the internal 16 stride memory bank structure of the CUDA shared memory. The extra rows allow threads watching the top and bottom rows of the board to still read valid data if they look "south" or "north" respectively. Zeros are empty positions, positive numbers are white stones and negative numbers are black stones. Numbers above 81 are reserved for special meanings in other modes.

The kernel steps through multiple modes in succession to play a game. It first analyzes the board to identify groups and the liberties of these groups. The first step is to assign each board position a number from 1 to 81 corresponding to its index in the board. This number is positive for white, negative for black, and zero otherwise. The algorithm then has each stone looking at its four neighbors for a lower number sharing its sign. If any are found, a flag is set and the stone assumes this number. This is repeated until no thread sets the flag. In this way, a connected group will assume the lowest number in the group.

The next step is to compute the liberties. Every thread watching an empty space examines its

neighbors. It stores the group numbers of its neighbors into a spot in a large array. This array is later reduced such that duplicates from the same empty space only add once. After this step, an array is created to hold the number of liberties associated with each numbered group.

It then takes the analyzed board and the active player to compute "good" moves for that player. Good moves are ones that have liberties of their own, capture enemy groups (play in the final liberty of that group), or at least don't reduce friendly groups to 2 libeties. Next it randomly selects a move from among the candidates and places a stone on that position. It then re-analyzes the board and removes killed stones. Killed stones belong to groups with no liberties. This is repeated until both players have no more "good" moves.

This kernel is configured to evaluate 128 blocks. Each block corresponds to a full simulation to game completion. A second kernel averages the output scores for the games and declares a score for the move using a prefix sum. This process is repeated for each valid move. Finally, the PC selects the highest scoring move to give to GNU Go.

This entire module was built into a library for use with GNU Go. GNU Go provides the GUI, and the library provides a move. Some glue code in the library converts a GNU Go board state into a state suitable for the CUDA kernels.

4 Performance

Evaluating the performance of the Go game proved to be difficult. The kernel can evaluate one move in about .1 seconds. This corresponds to solving a board position every 4 milliseconds. This is reasonably fast, but it is not nearly as fast as the reference implementation of GNU Go. This is likely because the algorithm in GNU Go is more intelligent and less brute-force.

Using CGoban, a front-end for GNU Go and other Go utilities, I was able to compare the strength of the two engines. Unfortunately, the GNU Go engine performs better and wins against the CUDA-enabled engine. There are numerous reasons for this.

The main problem is exploiting parallelism in the algorithm. On a normal parallel computer, each core is free to execute code and branch independently of the others. This is not true on the GPU. The GPU is a SIMD machine, and it is forced to serialize operations when threads take different execution paths. This will happen if, for example, one thread is watching an unoccupied space and another thread is watching a space with a stone on it. At most steps of the algorithm, these threads will perform very different actions. Ensuring that these threads do not diverge wildly has been a problem. The GPU was tuned to perform high speed computation, but its conditional branching instructions are relatively slow as are its memory access routines. This application requires a lot of memory reading and conditioning further actions on what is read. There are effectively no floating point operations outside of the final averaging kernel, so the main strength of the GPU cannot be used. A more arithmetically intensive application or algorithm would map better.

Another problem has been getting randomness on the GPU. CUDA does not allow calls to the "rand" function inside the GPU device code, so another scheme for random number generation is needed. The CUDA SDK includes an implementation of the Mersenne Twister algorithm seeded by the CPU. It fills a section of GPU memory with random floating point numbers. This works well, but it involves some overhead to produce random numbers on the GPU.

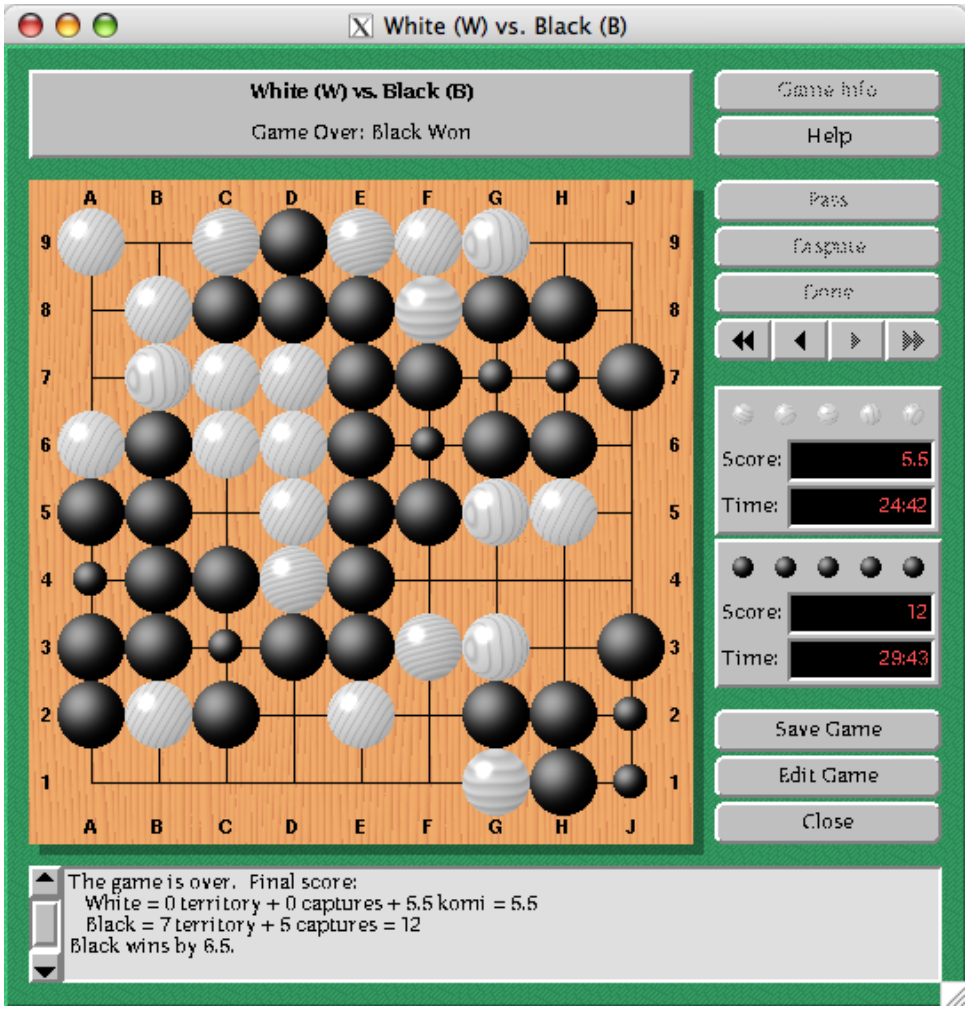

Figure 1: Sample game on CGoban. White is CUDA, black is standard

5 Results

Overall, the project was a success. However, some problems did arise with the graphics card because this problem maps poorly to the CUDA architecture. The first big problem is that this is a memory and logic bounded problem. There is relatively little arithmetic involved in playing a Go game in this manner. Most of the steps of the algorithm involve examining the neighbors of a space and choosing some action for that space based on the number of friendly or enemy stones. This utilizes a memory fetch and a test. Mathematical operations are not done on the results. This problem was improved by caching all data in thread shared memory, but it still limits performance.

The second major problem is the inherent divergence of the algorithm. This was addressed by formatting the data in an optimal way for fetching. It was also addressed by having many threads compute on dummy data to delay the branch. Later a simple test would determine whether the computed data should actually be used.

6 Running the code

In Evolution under the home directory for zacharyc are two executables. They are "cudago" and "gnugo". Gnugo is the stock GNU Go program compiled on Evolution. Cudago is the modified version compiled on Evolution for its GPU. This binary is hard linked to a file in zacharyc's home directory as well as to the current installation directory for CUDA on Evolution. Either program can be launched with the command line arguments "–monte-carlo –boardsize 9". This will enable the CUDA algorithm on "cudago" and give the comparable implementation on "gnugo". This worked as of last week, but requires exclusive access to the GPUs. After the Monday update this seems to be a problem, or usage of Evolution by other users may be a problem. CUDA has no mechanism for time-slicing access to the GPU devices.

The source code for this project is presently in zacharyc's home directory under "cudagame". The code compiles to a static library meant to be included into GNU Go as a binary blob. "compile.sh" performs this operation on Evolution. The output library can be copied into the "modgnugo-3.8/engine" directory for inclusion into the modified GNU Go. Because CUDA is installed into a nonstandard place, the library search path must be added to the executable at link time. Typing "setenv LD_RUN_PATH /usr/local/cuda/lib" before typing "make" takes care of this. The executable can then be found in the "interface" subdirectory of modgnugo-3.8. The only changes to gnugo-3.8 are in "genmove.c" and simply call the library routine.