Parallelizing Sudoku solver on GPU

Wang, Weiyi
ww1420@nyu.edu

Chun, Sungwoo
sc7408@nyu.edu

December 18, 2019

Abstract

Sudoku is a famous logical combinatorial puzzle game. It is an NP-complete problem, and does not have a good and fast algorithm for general problems yet. In the project, we did some survey on state-of-art Sudoku solver CUDA implementation, made some improvement to algorithm scalability and stability, and measured their performance.

1 Introduction

Sudoku is a famous logical combinatorial number placement puzzle. It is known to be an NP-complete problem, which means there is no a good and fast algorithm solve general Sudoku puzzles. In this project, we made a survey on state-of-art Sudoku solver CUDA implementation, made some improvement to algorithm scalability and stability, and measured their performance. Without restricting ourselves on traditional 9×9 Sudoku board, we define a rank N Sudoku game as follows: a square board with width N^2 is partitioned into N^2 sub-squares with width N for each. Some integers from $\{1, 2, \ldots, N^2\}$ are partially filled, and the player is required to fill the rest cells with integers from the same range, such that each row, each column, and each sub-square contains all numbers from $\{1, 2, \ldots, N^2\}$ (and as implied, without duplication). We call the ratio of initially filled cells to the number of total cells as "filling rate" r. A Sudoku board can have multiple solutions, and the number of solutions can easily explode for a less constrained initial condition. Since there is no algorithm to find out if a Sudoku board has a unique solution, we focus on algorithms that finds only one solutions from multiple ones.

2 Background: general approach and challenges

The general approach for solving a Sudoku board, as usually performed by humans as well, consists of two phases describes a tree-searching problem:

- Constraint propagation: when a cell is filled with a new number, it invalidates the option of the same number from other cells in the same row, column, or sub-square. This can potentially reduce the choice of other cells to one, allowing filling numbers in those cells with confidence, and generates another wave of propagation. The choices of a cell might also be reduced to zero, which indicates a contradiction and no solution is found for the board.
- Guessing and branching: when constraint propagation stagnates when there is no more cells that can filled with confidence, the player needs to pick a cell and assume one of its available options to enable more constraint propagation. The assumption may be wrong, which would results in a contradiction in its following constraint propagation, and in which case the player must step back to pick a different choice. We chose the numbers from a cell with least number of possible numbers left because it has higher probability of choosing the correct answer. Guessing and branching can, and likely will in hard problems, happen recursively, where the player has to perform multiple guessing before reaching a solution / contradiction.

2.1 Problem complexity

Based on experience of human players, it can be seen that the Guessing and branching phase is what determines the difficulty. When deeper branching is required, the complexity grows exponentially. The total number of cells is $O(N^4)$. For a hard problem where a sparse initial board is given, many of these cells has a potential to branch, so an conservative estimation of the problem complexity is $O(2^{N^4})$, assuming each branching only has two option to choose. In fact, a branching cell can have up to N^2 options, so the complexity can potentially goes to $O(N^N)$. This heavily stresses all tree-searching like algorithms on their scalability.

2.2 Performance instability

Another observation on solver performance is that the run time varies in a very large range. Experiments with rank 3 to rank 5 random Sudoku boards show that boards

with filling rate $r \geq 50\%$ are generally "easy": few to none branching ever happens, and a mediocre sequential solver can solve them in less than 0.1 second.

On the other hand, there is a barrier around $r = 40\%$, and any filling rate less than this would render the problem very "hard": branching layers increases abruptly, and it can take the program minutes to hours to solve a problem.

Another interesting observation is that if the filling rate gets below 20%, it is easier to solve the problem. This is due to the fact many empty cells makes many possible combination which mean that different assumptions can lead to different solutions.

Further more, even for the same problem size and filling rate, the run time can still vary a lot between seconds to hours. The search space and the solutions space implied by the initial condition can be very small or large, and it is unknown until one searches the tree.

Both the complexity and the instability make it much harder to characterize the general performance of a Sudoku solver. In this project, we focus on those "harder" problems with filling rate between 20% and 40%.

2.3 Parallelism friendliness

When the general approach is described as tree-searching, we can think of parallelizing by doing a breadth-first search on every branches. However, due to the fact that Sudoku is an NP-complete problem and it has 4.62×10^{38} possibilities for a 9×9 puzzle, no computer in the whole world can parallelize a single Sudoku problem. Also, serial version was implemented using recursion to simplify the backtracking, it needed different approach to parallelize in CUDA because maximum nesting depth of dynamic parallelism is limited to 24.

3 Literature survey

Apart from the traditional tree-search approach, there are also entirely different approaches that have been researched, such as simulated annealing[2] and genetic algorithm[3]. These methods are already well-developed and have their own performance characteristics, which is difficult to compare with a tree-search algorithm. In the project, we only focus on the tree-searching method and its variations, which can be a useful for a wide range of similar problems.

There is a recent CUDA implementation from a Caltech project[4]. It only supports the standard 9×9 board. We will generalize it to support larger size, as well as providing an alternative GPU algorithm.

4 Proposed implementation

Several variation of tree searching algorithm on CPU and GPU were attempted and measured. We will discuss some significant result from them below

4.1 Sequential DFS and DFS in general

As an initial implementation of sequential depth-first search was made. It follows strictly the two-phase searching described in Section 2. In this implementation, we discovered some general optimization technique that can also be applied to parallel program:

- Storing boards as bit flags: each cell can be represent as a bit string, each bit of which indicates whether an option of number is available. This compacts the storing space, and make constraint propagation as a sequence of bit operations.
- Heuristic branching: when branching is required, multiple cells can be chosen as the branching point. Some of them can leads to a solution faster while other may leads to more layer of branching. There is no concrete theory on which cell is the best to branch on, but experiment shows that branching on the cell that has the least amount of available options often yields a better result.

A simple parallelization attempt was made based on sequential DFS: while keeping the branching and searching structure, GPU parallel resource is deployed to solve constraint propagation cooperatively. This was designed with the idea that bit operations on different cells in constraint propagation can be done in parallel. However, the problem size of constraint propagation is never large enough for parallelization to gain benefit and overcome the overhead. This parallel algorithm attempt resulted an even slower program than the sequential version.

4.2 Parallel DFS

In the recent CUDA implementation in the Caltech project[4], their main idea was to dispatch thousands of independent CUDA threads and let each of them search down a branch until one of them finds a solution. The existing code only supports the traditional 9×9 Sudoku, so we reconstructed a generalized version for larger sizes based on the same idea.

A noticeable difference between their and ours is that they don't have the constraint propagation phase, and every cell is a mini branching point. This reduces the code complexity and memory usage by constraint propagation, with trade of missing optimization opportunity on a potentially long-burst constraint propagation sequence. We briefly measured the performance of their and our versions on 9×9 Sudoku, and they yield similar result.

The main problem of sequential DFS is that, even with heuristic branching, there are still chances that it would miss the correct branch and have to go back after several steps, resulting long run time. Parallel DFS

attempts to resolve this by covering a wide range of branches, and hoping that some of them could hit the correct branches.

4.3 DFS with parallel probing

Another idea of utilizing GPU parallel power is to enhance the heuristics in branching. In an article[5] of solvers for a different logic combinatorial puzzle, Nonogram, the author introduced a probing technique: when branching is required, instead of choosing one branching point and search down, choose multiple ones and search several steps ahead from them ("probing"). By comparing the intermediate result from these branching points, one can heuristically choose the most efficient one (so far) and continue on that branch. We applies the same technique on the Sudoku solver, and utilize GPU for parallel probing

5 Experimental Setup

We tested three programs (sequential DFS, parallel DFS, and DFS with parallel probing) on three machines (cuda2, cuda5, and a personal computer), with different sets of hard Sudoku problems (filling rate $r < 40\%$). We randomly generated Sudoku boards with fixed filling rate. However, as discussed in Section 2.2 and reported in *Solving Every Sudoku Puzzle*[1], fixed filling rate does not necessarily mean the same difficulty. In fact, the useful hard problems are really rare especially for lower rank. Because of these, we also test some known individual "hard" problems from *Solving Every Sudoku Puzzle*.

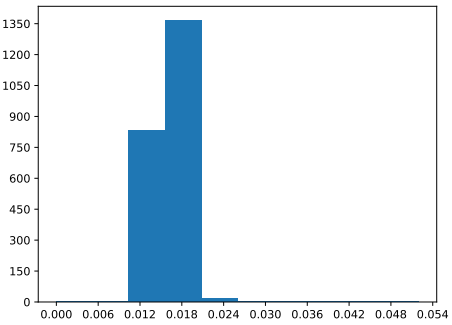
To generate random boards with a specific filling rate, we first randomly put a few numbers (filling rate $r = 10\%$) on the board without making contradiction, and "solved" the board into a finished Sudoku board, and then randomly removed some numbers to reach the desired filling rate. In general, generating a Sudoku board is at the same difficulty level as solving a Sudoku board, and we just used the same solver program to generate boards.

6 Result and discussion

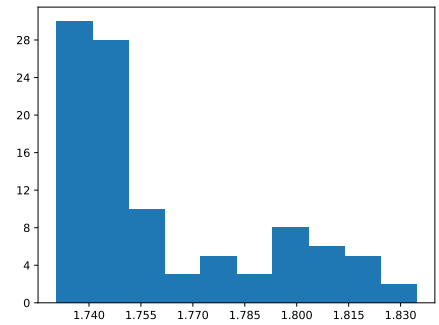
6.1 General observation

Performance on cuda2 and cuda5 servers fluctuates a lot due to heavy server load. In general, three machines shows similar performance characteristics for solver programs. We will just present the result from one machine to simplify the discussion.

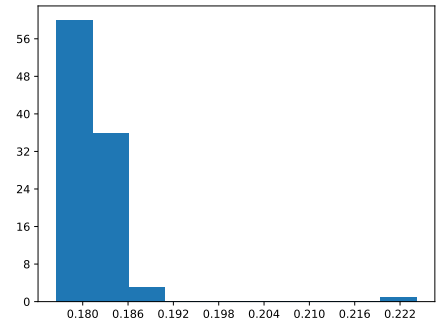
6.2 Rank 3



(a) Sequential DFS



(b) Parallel DFS



(c) DFS with parallel probing

Figure 1: $N = 3$, $r = 20\%$

Run time distribution for random $N = 3$, $r = 20\%$ problems are plotted in Figure 1. It can be seen the sequential version completely outruns in this cases. Further investigation shows that these randomly generated problems shows that most of the easy Sudoku problems can be done by branching less than 10 times and all the 1st attempt on branching succeeded without any backtracking. Which means that it just ran sequentially to the end. For these easy problems, which takes less than 1ms, sequential version already finished while we are launching CUDA kernels.

DFS with probing performs better than parallel DFS in this case. Parallel DFS split too much resource in searching in breadth on problems which could have been one-shot on one branches.

Table 1: A hard board

					6			
	5	9						8
2					8			
	4	5						
		3						
		6			3		5	4
			3	2	5			6

6.3 Rank 3 - specific hard problem

We also tested some known hard rank 3 Sudoku boards. For example, one of them is a board with 17 initial clues ($r = 21\%$) from *Solving Every Sudoku Puzzle*, as shown in Table 1.

The run time result for three programs are

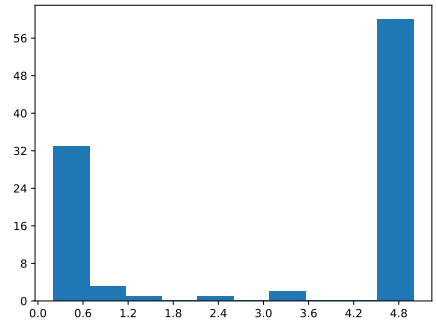
- Sequential DFS: 7.5 seconds.
- Parallel DFS: 1.7 seconds.
- DFS with parallel branching: 0.2 seconds.

It can be seen that the sequential DFS spikes up its run time a lot comparing its usual run time on random problems, while the two GPU versions remain in their typical run time range. This problem had 658,725 branching compared to 8 for easy problem. This shows that the GPU versions have more stable performance on hard problems. Other specific hard problems also show similar behavior.

6.4 Rank 5

Run time distributions on randomly generated rank 5 boards display long-tail property: for the same filling rate, many are solved in less than 1 seconds, while others run for nearly an hour or even more. This shows the performance instability as discussed in Section 2.2.

We set a 5-second timeout on all tests to avoid waiting for long time for those problems that takes hours to solve. In the histogram displayed below, the right-most column represents the amount of timeout.



(a) DFS with parallel probing

Figure 2: $N = 5$, $r = 10\%$

Figure 2 shows the run time distribution for DFS with parallel probing on rank 5 $r = 10\%$. The timeout rate is about 60%. What is not shown is the distribution of parallel DFS, which actually has a timeout rate of 100%. The search space of rank 5 problem is so large that thousands searching threads in parallel DFS is not large enough to cover the correct branch, and it shows worse performance than heuristic probing.

6.5 Rank 6 and up

None of test programs can consistently ($> 5\%$ probability) finish a rank 6 hard problem in a reasonable time.

6.6 Detailed profiling

Profiling with 'nvprof' shows that both GPU programs spend most time on the tree searching kernel ($> 90\%$). For DFS with probing, another important run time contribution happens in finding the branching point with least options, which takes 5% time for rank 3 and 10% time for rank 5. Finding branching points might also be a big time consumer for parallel DFS, but it is not a separate kernel there to be logged. The branching point searching is for better heuristics, but as it takes much time, a faster heuristic might be better to reduce the slow down.

Parallel DFS displays a large time consumption on in-device memory transfer, which is 2% for rank 3 and 5% for rank 5. The large memory transfer in the implementation is a compensation to lack of thread synchronization, which is because each thread works on its own branch independently, but which also prevents in-place memory operation without threads corrupting each other's data. A better synchronization mechanism might be deployed to reduce the memory transfer.

6.7 Scalability and extensibility

As discussed in Section 2.1, the Sudoku problem itself scales up the complexity really fast. While technically

the same program architecture for all test programs can be used for larger sizes, the run time is impractical. Revolutionary new algorithm is needed for approaching the Sudoku problem with large size. Nevertheless, heuristic probing shows a little bit better scalability than parallel DFS in rank 5, and may have the potential to scale up to larger size with further research on improving the heuristic.

A restriction on scalability from doing constraint propagation in GPU shared memory is that the largest rank for which the shared memory can hold at least one bit-string board is 8, because $8^4 * 8^2/8 = 32KB < 48KB$. For larger problem size, the program needs to be modified to work on global memory, which would further reduce the performance.

Solver for other variation of Sudoku (different subregion shape etc.) and similar logic puzzles can be made available relatively easily by changing the coordinate iterating rule in constraint propagation. However they will still likely suffer from the same scalability problem. Exceptions are when there is a concrete theory on good heuristic branching method for specific board pattern and shape.

7 Conclusion and future work

We have generalized the existing parallel DFS to larger sizes, and also developed the parallel probing program which can performs better in some cases. We are still far away from good scalability. For future work, one may try combining parallel probing and parallel DFS together. While requiring more parallel resource, it may collectively results in even better heuristics.

Things we learned in this project, with some knowledge taught in the course reiterated:

- Problem size matters, and parallelization should focus on the large-size part. In Sudoku, the constraint propagation part never has a large size comparing to the branching, so focusing parallelization on constraint propagation doesn't have much benefit.
- Parallel search in DFS matters. Parallelism doesn't necessarily means BFS. By running multiple DFS in parallel, we have better chance of getting to the correct solution earlier. The parallelism in NP-complete problem is not processing more data but increasing the chances to get to the solution.
- For an NP-complete problem, problem space is huge and contain from very easy or hard problems. Instead of simply taking an average run time, a distribution plotting and careful examination on individual problems can be more helpful.

Supporting material

Source code for sequential DFS, parallel DFS and DFS with probing are provided. See `README.txt` for how to use them.

References

[1] Norvig, Peter, *Solving Every Sudoku Puzzle* `http://norvig.com/sudoku.html`.

[2] Karimi-Dehkordi, Zahra and Zamanifar, Kamran and Baraani-Dastjerdi, Ahmad and Ghasem-Aghaee, Nasser, *Sudoku Using Parallel Simulated Annealing*, (2010).

[3] Sato, Yuji and Hasegawa, Naohiro and Sato, Mikiko, *GPU acceleration for Sudoku solution with genetic operations*, (2011).

[4] Duan, Victor and Teng, Michael, *Parallelized Sudoku Solver on the GPU*, `https://github.com/vduan/parallel-sudoku-solver`

[5] Wolter, Jan, *Survey of Paint-by-Number Puzzle Solvers*, `https://webpbn.com/survey/index.html`