Sudoku Solver Acceleration with CUDA

Ivannia Gomez Moreno[2], Ulises Orozco-Rosas[1,], Kenia Picos[1]*

[1]CETYS Universidad, Av. CETYS Universidad No. 4. El Lago, C.P. 22210, Tijuana B.C., México
[2]University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093, United States
[ivgomezmoreno]@ucsd.edu, [ulises.orozco, kenia.picos]@cetys.mx

Abstract

The classic Sudoku is a puzzle with simple rules that can become complicated to solve automatically due to the exponential increase in possible solutions as the board size increases. This paper presents an algorithmic solution to solve any size Sudoku incorporating an implementation in CUDA C/C++ that, consequently, leverages parallel programming on a GPU. This involves using threads and processes with shared and distributed memory for various solution strategies. These solution strategies encompass backtracking, where multiple alternatives are tried until reaching a solution, and rule-based algorithms that rely on heuristics to solve Sudokus, among others. Regardless of the implementation approach, it is pertinent to conduct a comparison between the solution achieved with a parallel algorithm utilizing a GPU and a sequential algorithm processed on the CPU. This is done to quantify the performance of the solution. The proposed algorithm managed to accelerate the traditional backtracking approach by up to 7x on a typical 9x9 Sudoku. While getting 7x and 3x reduction on 16x16 and 25x25 boards respectively.

Resumen

El Sudoku clásico es un rompecabezas con reglas simples que pueden volverse complicadas de resolver automáticamente debido al aumento exponencial de posibles soluciones a medida que aumenta el tamaño del tablero. Este artículo presenta una solución algorítmica para resolver Sudokus de cualquier tamaño, incorporando una implementación en CUDA C/C++, que aprovecha la programación paralela en GPU. Esto implica el uso de hilos y procesos con memoria compartida y distribuida para diversas estrategias de solución. Estas estrategias de solución incluyen 'backtracking', donde se prueban múltiples alternativas hasta encontrar una solución, además de algoritmos basados en reglas que se apoyan en heurísticas para resolver Sudokus. Independientemente del enfoque de implementación, es pertinente realizar una comparación entre la solución del algoritmo paralelo utilizando GPU y un algoritmo secuencial procesado en CPU. Esto se hace con el objetivo de cuantificar el rendimiento para encontrar la solución. El algoritmo propuesto logró acelerar el enfoque tradicional de 'backtracking' hasta 7 veces en los Sudoku típicos de 9x9. Además, se obtuvo una reducción de 7 veces en tableros de 16x16 y de 3 veces en tableros de 25x25.

Keywords— *Sudoku problem, GPU acceleration, backtracking algorithm, heuristics, parallel programming*

1. INTRODUCTION

Classical Sudoku constitutes a mathematical puzzle wherein the objective is to fill a 9x9 grid with numbers from 1 to 9. The grid is subdivided into 3x3 sections, and based on the predetermined numbers, the remaining numbers must be placed with the sole restriction that no repetition of numbers is allowed in each row, column, or subdivision. One of the most employed programmed algorithms for solving Sudoku puzzles is backtracking. In this algorithm, the program systematically explores every potential solution until success; if unsuccessful, it backtracks to the previous step and explores an alternative. Despite its straightforward rules, the puzzle poses highly complex challenges as the initial pre-established numbers decrease. This complexity arises because the backtracking model involves testing an exponentially growing number of cases as the predetermined numbers diminish [1].

To accelerate this procedure, this paper introduces an algorithm designed to resolve Sudoku puzzles with parallel programming on a GPU, employing a backtracking approach. A primary challenge encountered in implementing backtracking within CUDA lies in the limitations imposed on the number of recursive calls the GPU can generate [2].

Consequently, implementing this solution on larger boards, where recursive calls may extend to significant depths, is not a straightforward task. It is crucial to acknowledge the myriad variations of Sudoku; however, the scope of this project is confined to solving classic Sudoku problems exclusively. The algorithm presented as a solution could serve as a catalyst for generating other solutions related to it, such as optimization problems, search algorithms, resource planning, and, in general, problems involving variables and constraints [3]. The main contributions of this paper are as follows:

- Develop an algorithm in CUDA C/C++ to parallelize the search for solutions to Sudokus of any size using backtracking and adhering to the basic rules of Sudoku.
- Conduct a time comparison for the solution search of an incomplete Sudoku board between a sequential algorithm on the CPU and a parallelized algorithm on the GPU. Our algorithm accelerated the traditional backtracking approach by up to 7x on a typical 9x9 Sudoku. While getting 7x and 3x reduction on 16x16 and 25x25 boards respectively.

The article is organized as follows: Section 2 begins with an introduction to key ideas in parallel computing. Section 3 then describes the implementation of the traditional sequential backtracking approach, followed by a presentation

* Corresponding author.

of the proposed GPU acceleration. Section 4 will present the experimental results, while Section 5 will conclude up the paper and summarize the main findings of this study.

2. BACKGROUND

Section 2.1 introduces key concepts for parallel computing in CUDA. Threads, processes, memory models, communication mechanisms, parallel algorithms, and programming are all covered in this section. Section 2.2 delves into Sudoku-solving methodologies, stressing backtracking and rule-based algorithms needed to execute the proposed solution, as well as the Boltzmann Machine.

2.1 Basic Concepts

We first introduce several basic concepts for the problem.

- **CUDA C/C++**: programming platform that facilitates the acceleration of computations by leveraging GPUs. It enables parallel programming to execute tasks simultaneously across multiple GPU cores.
- **Parallelization**: the process of breaking down a task into smaller subtasks that can be executed concurrently on different CPU or GPU cores. Parallelization can be utilized to enhance efficiency and reduce the execution time of algorithms.
- **Time comparison of solution search:** aims to evaluate the efficiency of both sequential and parallel algorithms. Execution time can be measured using profiling tools that analyze the runtime of different parts of the code. Time comparisons assist in determining which algorithm is faster and more effective in solving Sudoku problems.
- **Parallel architectures**: hardware systems designed to support parallel computing, such as clusters, grids, supercomputers, and multiprocessors.
- **Threads and Processes**: Threads and processes are basic units of work that can be executed simultaneously in a parallel system. Threads are sub-processes that share hardware resources, while processes are separate instances of a program that run independently [4].
- **Shared and Distributed Memory:** In shared memory systems, multiple processors share a single physical memory [5], whereas in distributed memory systems, each processor has its own physical memory.
- **Communication:** Essential in parallel computing to enable processors to exchange data and coordinate their work. Communication mechanisms include shared memory, distributed memory, and interconnection networks.
- **Parallel Algorithms:** Specifically designed algorithms to run on parallel systems. These algorithms divide tasks into independent components that can be executed simultaneously on different processors, significantly enhancing performance.
- **Parallel Programming:** Process of writing software to run on parallel systems. This involves dividing the work into independent tasks and coordinating communication and synchronization between processors.

2.2 Possible Solution Strategies

Multiple strategies have been developed to solve Sudoku puzzles. Next, we list the most common:

1. **Backtracking**: It is an exhaustive search algorithm used to find all possible solutions to a problem by systematically exploring all options. If a point is reached where further progress is not possible, the algorithm backtracks to a previous option and tries a different value [6]. This method ensures that a solution will be found if one exists. In the case of Sudoku, the backtracking algorithm seeks a solution following the game rules, trying different numbers in each cell until a combination satisfies all constraints. This is not a new idea; proposals for solving Sudokus using backtracking have been made since 2014 [7]. Backtracking is an algorithm with exponentially increasing execution time, a valid consideration for this case study as Sudoku is an NP-complete problem, and a polynomial solution is unknown [8].
2. **Rule-Based Algorithm**: It relies on heuristics to solve Sudokus, trying various rules that allow filling cells or eliminating candidate numbers [8]. The rules followed include:
    a. Naked Single: A cell with a single candidate number.
    b. Hidden Single: If a region (row, column, or box) has a cell as the only option for storing a specific number, that number must be placed there.
    c. Naked Pair: If a region contains two cells, each with the same two candidates, the rest of the region can exclude these candidate numbers.
    d. Hidden Pair: If a region contains two cells that can hold two specific candidates, these cells form a hidden pair, and any other candidate is excluded from that pair.
    e. Guessing: An empty cell is filled with one of its candidates. The Sudoku-solving process continues until it reaches a solution or an invalid state. In the latter case, the guessed number is replaced with another, and the process is repeated.
3. **Boltzmann Machine**: This algorithm models Sudoku using a neural network capable of solving constraints. Each puzzle is viewed as a "constraint" that describes which nodes cannot be connected to each other. These constraints are encoded as weights in a neural network, which solves them in a way that its active nodes indicate the chosen numbers [8].

This paper centers its attention on the backtracking solution, given its more straightforward implementation in parallel.

3. IMPLEMENTATION

The algorithm's execution time on the CPU is primarily influenced by the Sudoku's difficulty and the number of given clues. As the number of clues decreases, the solution space becomes larger, necessitating more time to explore each potential solution space.

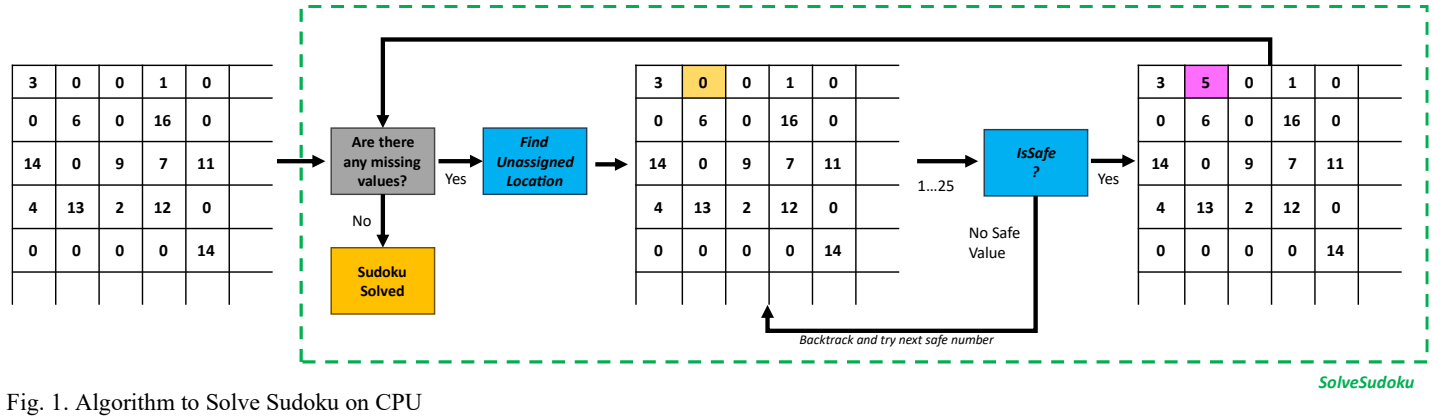


Fig. 1. Algorithm to Solve Sudoku on CPU

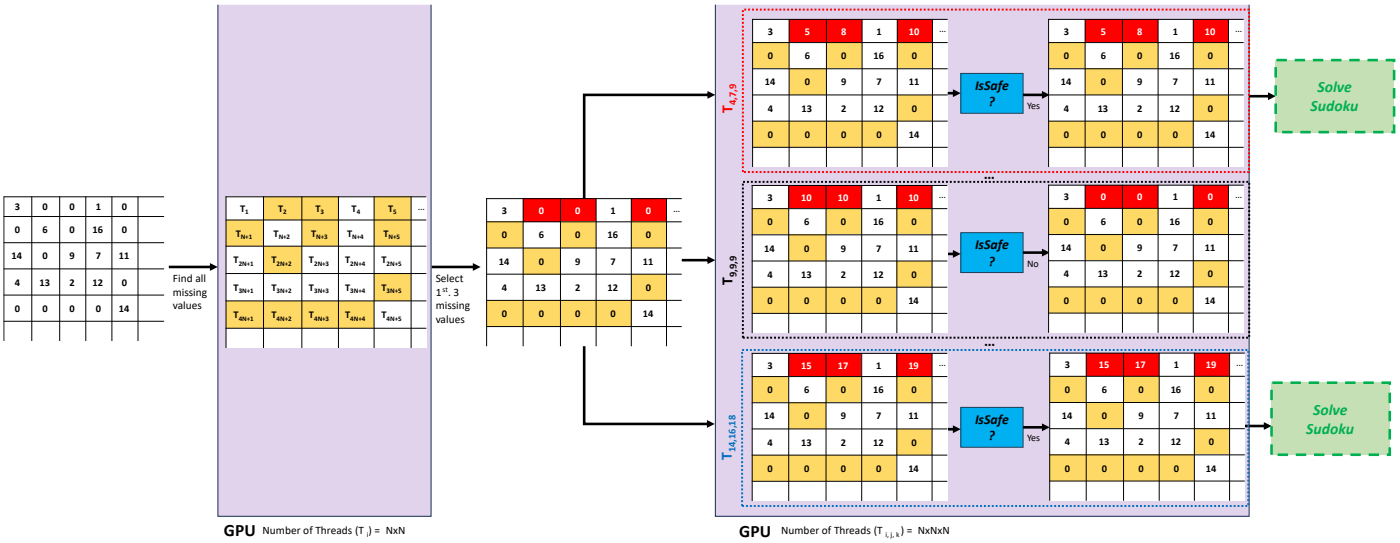


Fig. 2. Algorithm to Solve Sudoku on GPU

3.1 CPU

As previously mentioned, the backtracking strategy will be employed, leveraging the concept of recursion within the CPU algorithm. This approach will be applied to various 9x9 Sudokus, each solved ten times to obtain an average CPU solving time. Within the code, three main functions, in addition to the main function, are utilized as shown in Fig. 1:

- *FindUnassignedLocation*: This function, through a nested 'for' loop, examines each cell to identify the first number that has not yet been assigned. This serves as the starting point for developing a solution path.
- *isSafe*: It checks that the assigned number is not already present in any cells within the same row, column, and 3x3 subgrid. This is achieved through specific 'for' loops for each. Returns 'TRUE' only when the number proposed is not present in these three scenarios.
- *SolveSudoku*: This is the main algorithm responsible for backtracking. It starts with two empty pointers to the row and column locations (row and col, respectively) to be worked on. The *FindUnassignedLocation* function checks if any number is unassigned, changing the pointers of row and col to the first available location. If no unassigned location is found (all the cells are full), the Sudoku is solved, and *SolveSudoku* returns true, propagating the solution to all previous calls to the algorithm. The Sudoku is now solved in the same grid matrix, and it no longer explores other options. Otherwise, it enters a 'for' loop, checking all options (from 1 to 9) in that cell. If the *isSafe* function returns true, a tentative assignment of that number is made in that cell, and recursion proceeds with a call to *SolveSudoku*, repeating the process. When it reaches a point where the solutions taken so far are not viable, indicating that SolveSudoku returned false at some point, all tentative assignments are undone until another possible path is found.

3.2 GPU

Unlike the CPU, these Sudokus are only attempted to be solved twice, and the time is measured during the second iteration. This is done to exclude the time taken for the GPU connection. Two different functions were considered for parallelization as shown in Fig. 2:

- *FindUnassignedLocation*: As mentioned earlier, this function is called for each sub-level of recursion being worked on at the moment and involves a nested loop. To parallelize this function, blocks of 32 threads and the

necessary number of blocks per grid are considered. Simultaneously, all cells of the Sudoku are examined, and each thread saves its state in an array. This array is updated through a pointer to search for the next available empty space. This kernel accepts two parameter arrays: the initial incomplete board and a pointer to the array where the response will be stored. This kernel reduces the need for a double 'for' loop that looks through each column and row.

- *CreateTablesKernel*: This is a new kernel function that simultaneously generates various possible boards that adhere to Sudoku rules. This kernel starts with blocks of a single thread and NxNxN in the grid, where each block represents a possibility to fill the first 3 empty spaces on the board. For each thread, its blockIdx in each dimension is used to verify if it is a safe combination on the board, and its indices are saved in an array for use by the CPU. In this implementation, the `isSafe` function and its auxiliary functions were assigned values for both the host and the device, as mentioned in the research [9]. This way, the same function can be called from both the CPU and the GPU, each with its respective copy.

Once the kernel is executed, the CPU takes those validated values, copies them onto the board, and begins its recursion. If it does not find a solution, it only needs to choose the next solution; there is no need to revalidate because that was already done on the GPU. This eliminates the need for three 'for' loops for each cell to be checked (one for the column, another for the row, and another for the block) as it is done simultaneously at the outset.

4. RESULTS

Now, we show the results of the Sudoku Solver with CPU and the following acceleration with GPU. The metric to be used for comparing the two versions of the code will be the execution time in seconds. The first version will be a sequential code with backtracking processed on a CPU, exploring one possibility at a time. The second version will incorporate CUDA C/C++ to enable processing on the GPU. In the parallel algorithm, a check is performed to examine the first three empty squares. Once it finds a permutation that does not break the Sudoku rules, it is handed over to the sequential algorithm to fill in the remaining empty squares. While this strategy heavily depends on the sequential implementation, it is also essential to consider the limitations when generating CUDA code, because the GPU has a limit on the number of recursive calls that a thread can take with its limited memory.

The entire implementation will be carried out in the C++ language, and for parallelization, it will leverage CUDA libraries and be processed on an NVIDIA GPU, specifically an NVIDIA Tesla T4, and on an Intel(R) Xeon(R) CPU @ 2.20GHz.

The dataset used in the model implementation is from an online collection of Sudokus [10], with dimensions of 9x9. Specifically, those with a limited number of clues were chosen due to the significant increase in possible cases for the program, allowing for a clearer demonstration of the advantages of using the GPU. The data will be passed to the program as a matrix (a vector of vectors), where 0 represents a space that can be filled with a positive number between 1 and 9. Additionally, three 16x16 Sudokus of varying difficulty and three 25x25 Sudokus, featuring a much larger solution space, were used. It is noteworthy that all Sudokus in the dataset have the property of having a unique solution, thus avoiding ambiguity in obtaining the results.

4.1 CPU

The initial experiments were conducted using the CPU, and the following results were observed for the last fifty *9×9* Sudokus, specifically focusing on the most challenging ones to solve within this size. In Fig. 3(a), the microsecond time taken by the CPU to solve Sudoku is analyzed. It is observed that there is a significant increase in the time for certain Sudokus, taking up to 3 seconds to find the correct solution. Based on the objectives, it is anticipated that this time will be reduced with the optimization with CUDA using GPUs. Additionally, the backtracking code was tested with 16×16 Sudokus, and the results can be observed in Fig. 3(b). Similar to the previous graph, with fewer clues, there is a larger solution space, requiring more solutions to be examined before reaching the correct one. However, the longest execution time is observed in the case of the 25×25 Sudoku, as depicted in Fig. 3(c), where it can be observed that the time for solutions extended to seconds, indicating a significant opportunity for parallelization.

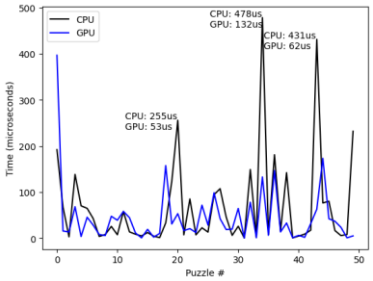
4.2 GPU

The comparative graph of resolution times for 9x9 Sudokus in Fig. 3(a) reveals that, overall, the implementation using the GPU outperforms the CPU, achieving a significant performance improvement. However, it's important to note that in some cases, the GPU's resolution speed was lower. This is because the problem size remains relatively small, and the data transfer process to the GPU generates some overhead [11]. Despite this, the GPU implementation achieved considerably faster times at three highlighted points on the graph. In these cases, the GPU solved Sudokus 4.8, 3.6, and 7 times faster than the CPU, demonstrating its ability to accelerate the resolution process in challenging situations.

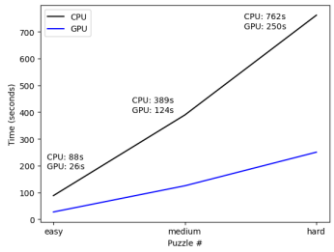
When comparing results with 16x16 Sudokus in Fig. 3(b), the advantage of using the GPU in terms of resolution speed is clear. At the easy level, the GPU managed to solve puzzles approximately 1.5 times faster than the CPU, with times of 10s compared to 15s, respectively. At the medium level, performance decreased minimally, solving the same problem in approximately 1.4 times less time than the CPU, with times of 43s compared to 62s. However, the most significant impact was observed at the difficult level, where the GPU was exceptionally faster, solving puzzles 7.25 times faster than the CPU, with times of 28s compared to 203s.

Finally, the results of the comparison between the CPU and GPU in the resolution of 25x25 Sudokus in Fig. 3(c) consistently show the outstanding superiority of the GPU. At the easy level, the GPU achieved an average time of 26s, which is approximately 3.4 times faster than the CPU, which

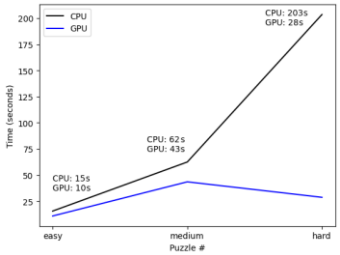
took an average of 88s. At the medium level, the GPU once again demonstrated its efficiency, taking an average of 124s, approximately 3.1 times faster than the CPU. At the difficult level, the GPU implementation took 250s to solve the puzzle, implying it was 3.0 times faster than the CPU, which required an average of 762s.



a) Execution time 9x9



b) Execution time 16x16



c) Execution time 25x25

Fig. 3. Execution times between CPU and GPU at different levels of difficulty

5. CONCLUSION

In this paper, we develop a parallel algorithm capable of accelerating the computation of a Sudoku solution. However, certain important details need to be highlighted. Firstly, the parallel model manages to achieve a shorter execution time than the sequential algorithm in cases where a more profound search in terms of possible solutions is required. This is attributed to the structure of the parallel algorithm, specifically the section that is parallelized, which is the search for possible solutions. Therefore, if the solution is found within the initial iterations, the difference between the CPU and GPU implementations is not substantial. This solution underscores the complexity of identifying the best implementation in parallel algorithms, as numerous variables, such as thread and block configurations, memory access frequency, and the type of memory used, can influence execution time.

Hence, it is affirmed that the implemented parallel algorithm efficiently solves a Sudoku when the search for possible solutions becomes extensive. Nevertheless, this parallel algorithm can be further optimized by leveraging shared memory, thread utilization, and expanding parallelized sections, among other techniques.

Despite the apparent simplicity of our solution, optimizing GPU usage through backtracking proves to be a powerful technique with applications across various industrial domains. Its ability to accelerate intensive data processing significantly reduces wait times in computational optimization tasks. In bioinformatics, this strategy facilitates the comparison and alignment of DNA and protein sequences, while in logistics and transportation, it enhances the search for optimal routes in complex systems. Additionally, resource optimization enables efficient allocation in linear programming problems, such as production planning. Accelerating these processes through the efficient management of threads on GPUs remains an innovative area with great potential for automating tasks that traditionally require manual and computationally expensive searches

6. BIBLIOGRAPHY

[1] Arteaga, A., Orozco-Rosas, U., Montiel, O., Castillo, O. (2022). Evaluation and Comparison of Brute-Force Search and Constrained Optimization Algorithms to Solve the N-Queens Problem. In: Castillo, O., Melin, P. (eds) New Perspectives on Hybrid Intelligent System Design based on Fuzzy Logic, Neural Networks and Metaheuristics. Studies in Computational Intelligence, vol 1050. Springer, Cham.

[2] Orozco-Rosas, U., Montiel, O., Sepúlveda, R. (2015). Parallel Evolutionary Artificial Potential Field for Path Planning-An Implementation on GPU. In: Melin, P., Castillo, O., Kacprzyk, J. (eds) Design of Intelligent Systems Based on Fuzzy Logic, Neural Networks and Nature-Inspired Optimization. Studies in Computational Intelligence, vol 601. Springer, Cham.

[3] Orozco-Rosas, U., Picos, K., Montiel, O., Castillo, O. (2021). GPU Accelerated Membrane Evolutionary Artificial Potential Field for Mobile Robot Path Planning. In: Castillo, O., Melin, P. (eds) Fuzzy Logic Hybrid Extensions of Neural and Optimization Algorithms: Theory and Applications. Studies in Computational Intelligence, vol 940. Springer, Cham.

[4] S. Cook, CUDA programming: a developer's guide to parallel computing with GPUs. Newnes, 2012.

[5] M. G. John Cheng and T. McKercher, Professional CUDA C Programming. John Wiley Sons, Incorporated, 2014.

[6] F. N. Abu-Khzam, K. Daudjee, A. E. Mouawad, and N. Nishimura, "On scalable parallel recursive backtracking," Journal of Parallel and Distributed Computing, vol. 84, pp. 65–75, 2015

[7] A. K. Maji and R. K. Pal, "Sudoku solver using minigrid based back-tracking," in 2014 IEEE International Advance Computing Conference (IACC). IEEE, 2014, pp. 36–44

[8] P. Berggren and D. Nilsson, "A study of sudoku solving algorithms," Master's thesis, KTH Computer Science and Communication, Stockholm, Sweden, 2012.

[9] R. Farber, CUDA application design and development. Elsevier, 2011.

[10] S. Qiu, X. Shu, Y. Xie, and Y. Wang. (2018) Parallelized giant sudoku solver. [Source code]. https://github.com/shiyunqiu/CS205 Sudoku

[11] Orozco-Rosas, U., Montiel, O., Sepúlveda, R. (2017). An Optimized GPU Implementation for a Path Planning Algorithm Based on Parallel Pseudo-bacterial Potential Field. In: Melin, P., Castillo, O., Kacprzyk, J. (eds) Nature-Inspired Design of Hybrid Intelligent Systems. Studies in Computational Intelligence, vol 667. Springer, Cham.