Parallel Sudoku Solver on GPU

Ziyun Wang

Abstract

GPGPUs are nowadays widely used to accelerate programs with heavy computations, and is especially suitable for independent data. Backtracking is one of the classic search algorithms which is computationally expensive, but with serious data dependency. To study how backtracking algorithms can be accelerated with GPGPUs, this project selected a typical application of backtracking, Sudoku solving. In this project, three different parallel algorithms are proposed and evaluated. From the experiments, parallel methods can outperform sequential methods on difficult puzzles with carefully tuned parameters. The experiments also reveal that the Dancing Links data structure can speedup algorithms on CPU, but is not suitable for GPU.

1 Introduction and Related Works

Sudoku[1] is a well-known puzzle played on a $N \times N$ (9×9 in most cases) grid. Given a partially filled board with some empty slots, the objective is to fill it with digits $1 - N$ so that each row, each column, and each $n \times n$ box ($n^2 = N$) contain exactly digit $1 - N$ once. Figure 1 provide an example of solving a Sudoku puzzle.

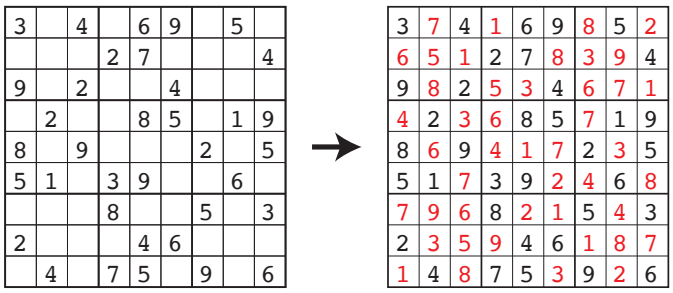

Figure 1: An example of a Sudoku puzzle and its solution.

While Sudoku serves as an interesting puzzle for humans, there are also a number of algorithms proposed for solving Sudoku with computer. The simplest one is backtracking[2], which is a brute-force search method based on depth-first search strategy. This algorithm is simple, easy to implement, and solution is guaranteed on valid Sudokus. However, its worst time complexity is high, and this method can be extremely slow sometimes. An optimized backtracking algorithm models the puzzle as an exact cover problem[3], which is a task of given a binary matrix, finding a subset of rows that each column has exactly one element of 1, in other words, each column is covered exactly once. Though this task is NP-complete [5], Knuth proposed an elegant data structure called Dancing Links [6] that can efficiently solve it. Details of this method is provided in section 2.2.

There also exist some algorithms based on methods other than backtracking. One direction is to use stochastic optimization approaches, such as Genetic Algorithms [9], Simulated Annealing [9], Harmony Search [4], Artificial Bee Colony [8] etc. Stochastic methods are often used to solve NP-complete problems as they can provide a good approximation in a relatively short time. However,

[1] https://en.wikipedia.org/wiki/Sudoku
[2] https://en.wikipedia.org/wiki/Sudoku_solving_algorithms#Backtracking
[3] https://en.wikipedia.org/wiki/Exact_cover

these methods do not guarantee to find the exact solution. Other researchers also tried to model Sudoku as a constraint satisfaction problem and solve it with complicated reasoning algorithms [12]. In this project we do not study this approach in details.

With the prosperity of General Purpose Graphics Processing Units (GPGPUs), many computational intensive tasks are being solved with the powerful parallelism of GPGPUs. While there are not many research directly on migrating Sudoku solving algorithms to GPUs, many efforts have been devoted to optimizing backtrack algorithms with GPUs. Since backtrack can be viewed as the process of expanding a search tree, whose root node is the initial problem to be solved, and nodes with larger depths are more restricted, and have smaller scales. A widely-used strategy for GPU backtracking consists of two steps: an initial search on CPU and a parallel tree-based backtracking on GPU [2, 7, 11, 3]. The search on CPU expand search tree to some predefined depth or node size, and store all leaf nodes in an active set. The GPU search code will execute on the nodes in the active set in parallel. Such strategy performs well in regular scenarios, but it may suffer from load imbalance, and sometimes it may degenerate to be slower than a sequential implementation.

In recent years, with the support of CUDA Dynamic Parallelism (CDP), some CDP-based backtracking strategies are proposed to solve the irregular cases [10]. However, according to Pessoa *et al.* [3], these CDP-based strategies can be better in general, but are slower than the normal two-step strategy with well-tuned parameters. Dynamic parallelism may introduce overhead from stream creation and destruction, and kernel launches. Also, the use of CDP brings a lot more difficulties in coding and debugging.

So in this project, I will focus on parallelizing backtracking algorithm with the traditional two-step strategy. I first implement a simple version which directly migrates the sequential version to GPU. Then I also parallelize a core operation in Sudoku, which is the step of validating the puzzle result. I also explore the execution of Dancing-Links-based algorithm in parallel, and analyze the drawbacks of this algorithm on GPUs.

The rest of this report is organized as following. In section 2 and 3, I will introduce all the methods I implement in detail. Then section 4, shows the experiment results and analysis. Finally section 5 provides the conclusion.

2 Sequential Approaches

In this section, I will introduce the sequential methods I implement in this project, which include a brute-force backtracking and a Dancing Links based algorithm.

2.1 Brute Force

The baseline backtrack algorithm simply traverses all grids from the top left one to the bottom right one. For each empty grid, it attempts digit $1 - N$, and see if this number is valid given the partially filled board. If so, then it writes the digit on the board, and proceed to the next grid. If all digits are invalid, then it backtracks to the last grid that it attempts, and continue to try other digits. If the puzzle has an answer, then this algorithm will finish when all the grids are filled. Otherwise it will try out every possibility and returns.

2.2 Dancing Links

Modeling Sudoku as an Exact Cover Problem Consider the following binary matrix (example from [6])

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{1}$$

Algorithm 1 Algorithm X for solving exact cover problem

```
procedure EXACT_COVER_X(M, S)                    ▷ Input matrix M, set of chosen rows S
    for X in all rows of M do
        Remove all columns in M covered by row X
        Remove all rows in M in conflict with row X
        if M is empty then
            if All elements in X are 1s then                    ▷ Exact cover is found
                ans ← S, halt
            else                                                ▷ Current selection is wrong
                continue
            end if
        end if
        EXACT_COVER_X(M, S ∪ {X})
        Restore columns and rows of M
    end for
end procedure
```

We say rows 1, 4 and 5 form an exact cover of this matrix, because this set of rows have exactly one 1 in each column. The exact cover problem is to find such a set that exactly "covers" the matrix.

Sudoku can be modeled as an exact cover problem [1] with a simple translation. Take a 9×9 Sudoku as an example. First we treat each column of the matrix as a constraint.

1. For $1 \leq N \leq 81$, column N represents a digit is filled at grid (X, Y), where $N = (X-1)\times 9+Y$.
2. For $82 \leq N \leq 162$, column N represents digit D is filled at row X, where $N = (X-1) \times 9 + D + 81$.
3. For $163 \leq N \leq 243$, column N represents digit D is filled at column Y, where $N = (Y-1) \times 9 + D + 162$.
4. For $244 \leq N \leq 324$, column N represents digit D is filled in box B, where $N = (B-1) \times 9 + D + 243$.

Then we define the rows of the matrix. Each row will have four 1s. For each grid (X, Y), suppose it is in the box B, if the grid already has a digit D, then according to our definition of columns, the four columns that have 1 are $(X-1) \times 9 + Y$, $(X-1) \times 9 + D + 81$, $(Y-1) \times 9 + D + 162$, $(B-1) \times 9 + D + 243$. For example, if 7 is in grid $(4, 2)$, then the matrix will have a row with 1s at column $29, 115, 178, 277$, and 0s at other columns. If the grid is empty, then we insert 9 different rows following the same rule, with $D = 1, 2, \cdots, 9$. This means the grid can take any digit from 1 to 9.

Recall that for the exact cover problem, each column can only be covered once, therefore the rule of Sudoku can be fulfilled when solving exact cover problem.

Solving Exact Cover with Dancing Links A typical backtracking algorithm (Algorithm X) for solving exact cover problem is described in algorithm 1, which recursively removes columns and rows, and solve the problem of smaller scale. For example, if row 2 is chosen in this matrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{2}$$

column 1, 4, 7 are covered, and row 4, 5, 6 are in conflicts with row 2. We can remove them and solve a smaller problem with the following matrix

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \tag{3}$$

Algorithm 2 Cover and Uncover a column in Algorithm DLX

```
1: procedure REMOVE(c)                                  ▷ Remove a column from dancing links
2:     L[R[c]] ← L[c], R[L[c]] ← R[c]
3:     for all i ← D[c], D[D[c]], ··· , i ≠ c do
4:         for all j ← R[i], R[R[i]], ··· , j ≠ i do
5:             U[D[j]] ← U[j], D[U[j]] ← D[j]
6:         end for
7:     end for
8: end procedure
9: procedure RESTORE(c)                                 ▷ Restore a column from dancing links
10:    for all i ← D[c], D[D[c]], ··· , i ≠ c do
11:        for all j ← R[i], R[R[i]], ··· , j ≠ i do
12:            U[D[j]] ← j, D[U[j]] ← j
13:        end for
14:    end for
15:    L[R[c]] ← c, R[L[c]] ← c
16: end procedure
```

However, this smaller problem does not have an answer. So we should backtrack to the previous step, restore all the removed columns and rows, and try with another row.

Algorithm 3 Algorithm DLX for solving exact cover problem

```
1: procedure EXACT_COVER_DLX(o, S)                      ▷ Root node o, set of chosen rows S
2:     if R[o] = o then                                 ▷ The matrix is empty. Exact cover is found.
3:         ans ← S, halt
4:     else
5:         t ← R[o]                                     ▷ The top left element
6:         for all n ← D[t], D[D[t]], ··· , n ≠ t do
7:             for all j ← R[n], R[R[n]], ··· , j ≠ n do
8:                 REMOVE(j)                            ▷ remove corresponding columns and rows
9:             end for
10:            X ← n.row
11:            EXACT_COVER_DLX(o, S ∪ {X})
12:            for all j ← R[n], R[R[n]], ··· , j ≠ n do
13:                RESTORE(j)                           ▷ restore corresponding columns and rows
14:            end for
15:        end for
16:    end if
17: end procedure
```

Implementing this algorithm with data structure such as arrays, deleting and inserting an element in the middle is computationally expensive. Dancing Links is very powerful for such situation. It represents all 1s in the matrix with a four-way-linked list. Besides, it has a special root node and each column has a column indicator node. Each node n has 5 pointers, named $L[n], R[n], U[n], D[n], C[n]$, pointing to the node on its left, right, up, down, and the column indicator. Figure 2 shows the dancing links for the above example.

Removing and restoring rows and columns can be achieved efficiently with dancing links. Algorithm 2 provide the procedures used for removing and restoring a column. And the full backtrack algorithm with dancing links (algorithm DLX) is shown in algorithm 3.

3 Parallel Approaches

In this section, I will introduce how I re-implement the above two methods with CUDA. The general pattern of parallelizing a backtrack algorithm is to first construct a seed set of mutual exclusive search trees, and continue search on each tree on an independent thread. Algorithm 4 describes this process in pseudo code, where expand size s is tunable parameter controlling how

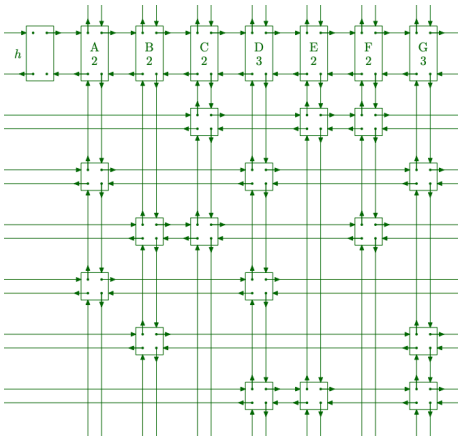

Figure 2: Structure of dancing links solving exact cover problem of matrix 2. Adapted from [6]. The columns are named A,B,C,D,E,F, and G. The number in each column indicator shows how many nodes are in that column.

many search trees are returned by initial search on CPU.

Algorithm 4 CPU-GPU Search Process

```
1: procedure CPU_GPU_Search(T, s)          ▷ Task to be solved T, CPU expand size s
2:     S ← expand_search_tree_cpu(T, s)
3:     Allocate data on GPU
4:     Transfer data to GPU
5:     gpu_search<<<#blk, #td>>>(S)        ▷ Launch GPU Search Kernel
6:     Transfer data back to CPU
7:     ans ← check_results_from_gpu( )
8: end procedure
```

3.1 Simple Parallel Brute Force

To parallelize brute force search, we need to separate the search process into two steps, an initial search on CPU and a parallel search on GPU. I implement the initial search with BFS instead of DFS, because it is easier to limit the size of expanded search trees with BFS, rather than set a maximum depth of DFS search.

For the search process on GPU, I implement two variations. The first one is a simple migration, where each thread search on an independent search tree. At the point view of each thread, this method is similar to the sequential version, except when an answer is found on a thread, it atomically modifies a global variable to notify other threads.

To get rid of the load of parameter passings during function calls, I do not use recursion when implementing this method. Instead, a stack is used to explicitly manage the backtrack process, which stores the grid location being filled. Since a Sudoku puzzle board is relatively small, the stack as well as the board itself can be stored in shared memory for fast access.

3.2 Optimized Parallel Brute Force

During the backtracking process, it takes a lot of time to check the validity of current number filled. Actually, different grids are independent and thus this step can be fully parallelized. In the second variation, I use a block of $N \times N$ threads to work on each search tree, and each thread is responsible for one grid. Thread $(0, 0)$ is denoted as leader thread, which controls the stack and the backtrack process. Only when finding which number is valid for current empty grid, all the threads work together.

In this method, the stack is also put in shared memory, while the board itself can actually be put in the register of each thread, providing even faster accesses. For each task group, $N + 4$ variables

Algorithm 5 Optimized Parallel Brute Force

```
1: procedure GPU_SEARCH_OPBF(S)                                  ▷ Set of search trees S
2:     Get board from S_blockIdx.x
3:     board_num ← board[threadIdx.x][threadIdx.y]                ▷ save board in register
4:     Declare stack, comm in shared memory                       ▷ comm is for in-block communication
5:     while answer not found do
6:         if this is leader thread then
7:             if answer already found then
8:                 atomically set global variable
9:                 comm[0] ← −1                                  ▷ notify other threads in the block to terminate
10:            end if
11:            Find the grid to try according to the stack
12:            comm[1] ← next number to try
13:        end if
14:        __syncthreads()
15:        if comm[0] = −1 then                                   ▷ for all threads in the block
16:            break
17:        end if
18:        For each n in [comm[1], N], save to comm if n is invalid
19:        __syncthreads()
20:        if this is leader thread then
21:            Find next valid number n
22:            if no such n found then
23:                Pop stack
24:                comm[2] ← −1 if the stack is empty             ▷ search failed, terminate
25:            else
26:                Push stack
27:            end if
28:            comm[3] updated ← digit at this grid
29:        end if
30:        __syncthreads()
31:        update board_num ← comm[3] of corresponding thread
32:        if comm[2] = −1 then                                   ▷ for all threads in the block
33:            break
34:        end if
35:    end while
36: end procedure
```

declared in shared memory are used for communication. Algorithm 5 describes this method in more details.

3.3 Parallel Dancing Links

I also try to parallelize the algorithm DLX, and implement it with CUDA using the same pipeline as in algorithm 4. Line 2 returns a set of several Dancing Links, and each thread on GPU works on one Dancing Links structure, follows the same algorithm as stated in algorithm 3.

Though Dancing Links is very efficient and powerful on CPU, it is not suitable to be implemented on GPU. There are several reasons.

First, for a $N \times N$ Sudoku puzzle, each grid correspond to at least 4 nodes, at most $4 \times N$ nodes. As the result, the size of a Dancing Links usually can be 20 or 30 times larger than a board itself. In this case, shared memory are too small to be used for optimizing the code. Besides, the algorithm cannot be largely parallelized, since GPU device memory is also quite small.

Second, dancing links is based on linked-list, which causes a terrible memory access pattern on GPUs. The memory access cannot be coalesced, and the latency of global memory accessing on GPU will be a bottleneck.

Third, the algorithm modifies the structure of Dancing Links frequently, and two structures between adjacent thread might be very much different. If threads in a warp are dealing with Dancing Links of very different shapes, this may lead to serious branch divergence issue. As a

result, in some situations, only activating one thread in each warp and launching more blocks will be faster than assigning task on each thread in a warp and launching less blocks.

4 Experiments and Analysis

4.1 Experiment setting

In the project, I implement five methods in total, including two sequential methods: Brute Force (BF), Dancing Links (DLX); and three parallel methods: Simple Parallel Brute Force (SPBF), Optimized Parallel Brute Force (OPBF), and Parallel Dancing Links (PDLX). These methods will be compared in the experiments on 7 different puzzles. These puzzles have different sizes, and different level of difficulties. Table 1 list all the puzzles and their properties.

Parallel methods have parameters to tune, such as expand size s, threads per block $\#td$. These parameters have great impact on the performance of each method. Next section will study this issue.

Puzzle #	1	2	3	4	5	6	7
Size	9×9	9×9	9×9	9×9	16×16	16×16	16×16
Difficulty	Easy	Medium	Hard	Invalid	Easy	Medium	Hard

Table 1: Properties of test puzzles.

4.2 Results analysis

The results of the experiment is provided in table 2. Note that puzzle 1 is so easy that it will be solved during initial CPU search if expand size is greater than 16, thus $s = 16$ is chosen for all parallel methods. From the table, sequential methods are very efficient in easy problems, while parallel methods are powerful when the puzzle becomes challenging. This is the same with our intuition.

Comparing the two sequential methods, Dancing Links is not always faster than Brute Force. The execution time of backtrack based algorithms depend much on the input data itself and the search order. If the search order of the algorithm is accidentally close to the correct answer, then the search process will finish in a short time.

Comparing the parallel methods, OPBF performs the best in general, and PDLX the worst. A detailed analysis on the parallel methods are provided in the following parts.

Parameter tuning For the parallel methods, the parameters play an important role, and greatly influence the performance. The execution time of same method on a same puzzle can be more than

Puzzle #	BF	DLX	SPBF	OPBF	PDLX
1	0.011	**0.010**	0.284 (16,16)	0.292 (16)	0.288 (16, 16)
2	0.357	**0.057**	0.400 (32768,64)	0.358 (2048)	0.531 (2048, 64)
3	**0.383**	0.493	0.596 (32768,64)	0.505 (2048)	0.930 (8192, 32)
4	1.114	0.815	1.813 (65536, 64)	**0.668 (8192)**	1.532 (8192, 32)
5	**0.160**	0.589	3.295 (1024, 16)	1.084 (16)	1.685 (1024, 64)
6	0.495	1.539	1.049 (2048, 16)	**0.422 (16)**	5.213 (2048, 64)
7	∼300	∼75	1.683 (8192, 16)	**1.445 (2048)**	1.892 (8192, 64)

Table 2: Execution time (s) of each method on each puzzle. Reported time is calculated as an average of 3 different runs. For parallel methods, the result of best parameter is reported, and the parameter for the result is also provided in the parenthesis. Two numbers are expand size s and number of threads in each block. The block size of OPBF method is $N \times N$, so it only has one parameter s.

s	1	8	16	256	4096	16384	65536
Time(s)	12.045	0.587	0.422	0.492	0.562	0.611	0.844

Table 3: Execution time of OPBF on puzzle 6 with different expand sizes.

4 times longer with a different parameter. One key parameter in this task is the expand size s, which controls the size of active set returned from initial search. A smaller s means a faster initial search, fewer blocks, a faster data transfer, however, it also means limited parallelism and possibly longer execution for each individual thread. Take puzzle 6 and method OPBF as an example, table 3 shows its performance under different parameters. When s is small, there is not enough parallelism, and executing sequential code on a GPU is much slower than on a CPU. Oppositely, when s is too large, the overhead of initial search, communication, and executing large number of blocks will also slow down the program. Table 4 provide more profiling results to prove this.

The other parameter, number of threads in a block, has a comparatively minor effect on the performance, and it is also limited by the problem scale. For method SPBF, when the puzzle size is 16×16, a block with more than 24 threads will run out of shared memory space, so the number of threads in a block should be no more than 24.

s	8	16	256	65536
Kernel time τ_1	237.39	153.69	185.83	360.24
Memcpy time τ_2	0.0018	0.0028	0.023	17.43

Table 4: Kernel execution time (ms), memory copy time from host to device (ms) of OPBF on puzzle 6 with different expand sizes, given by `nvprof`.

Comparing SPBF and OPBF SPBF and OPBF use the same underlying algorithm, and have the same order of expanding search tree. And it seems OPBF method has lower utilization, as it uses the whole block to work on one search tree, and only one thread is active when operating on stacks. However, OPBF has better performance. I perform a case study on puzzle 4. OPBF achieves its best performance when $s = 8192$. If I also set $s = 8192$ for SPBF, its execution time is 2.232s (for $\#td = 64$), which is much worse than OPBF.

OPBF has a faster validation step, which reduces a lot of execution time for searching on each individual tree. Also, OPBF achieves a higher occupancy (0.514), compared to SPBF (0.051). This is achieved by saving the board information in registers, which reduces half of the shared memory accesses.

Analysis of PDLX The performance of PDLX is pretty unsatisfactory. Some reasons are mentioned in section 3.3. Here I provide some profiling results in table 5 for better illustration. Since shared memory is not utilized, and the memory access is not coalesced in PDLX method, it has large number of global memory access transactions.

Method	SPBF	OPBF	PDLX
Occupancy	0.0469	0.6249	0.3333
# Memory Load	4.2×10^6	1.4×10^8	9.5×10^9
# Memory Store	256	256	3.7×10^9

Table 5: Achieved occupancy, number of global memory load transactions, and number of global memory store transactions of three parallel methods on puzzle 7 with expand size $s = 8192$ and number of threads in each block $\#td = 16$, given by `nvprof`.

5 Conclusion

In this project, I studied the topic of parallelizing Sudoku solving algorithms with CUDA. Two sequential backtracking algorithms, brute force and DLX, and three parallel algorithms based on them are implemented and evaluated. The parallel algorithms are based on a common strategy, which first performs an initial search on CPU and then continue search on the node of initial search tree in parallel on GPU. I also tried a method which optimizes the parallel brute force method, which performs the step of validating Sudoku answer in parallel, and such two level parallelization provides a faster algorithm. However, in the experiments I find that Dancing Links algorithm is not suitable for GPU. And although parallel methods outperform sequential version in many cases, their performance heavily rely on manually tuned parameters. In real situations, parameter tuning based on inputs may not be an available choice. Better strategies which can manage the load imbalance issue should be studied to further improve the stability and generality of such algorithms.

References

[1] R. Beezer. Solving sudoku efficiently with dancing links. `http://buzzard.ups.edu/talks/beezer-2010-stellenbosch-sudoku.pdf`, 2010.

[2] T. Carneiro, A. E. Muritiba, M. Negreiros, and G. A. L. de Campos. A new parallel schema for branch-and-bound algorithms using gpgpu. In *2011 23rd International Symposium on Computer Architecture and High Performance Computing*, pages 41–47. IEEE, 2011.

[3] T. Carneiro Pessoa, J. Gmys, F. H. de Carvalho Júnior, N. Melab, and D. Tuyttens. Gpu-accelerated backtracking using cuda dynamic parallelism. *Concurrency and Computation: Practice and Experience*, 30(9):e4374, 2018.

[4] Z. W. Geem. Harmony search algorithm for solving sudoku. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 371–378. Springer, 2007.

[5] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[6] D. E. Knuth. Dancing links. *arXiv preprint cs/0011047*, 2000.

[7] L. Li, H. Liu, H. Wang, T. Liu, and W. Li. A parallel algorithm for game tree search using gpgpu. *IEEE Transactions on Parallel & Distributed Systems*, (1):1–1, 2015.

[8] J. A. Pacurib, G. M. M. Seno, and J. P. T. Yusiong. Solving sudoku puzzles using improved artificial bee colony algorithm. In *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, pages 885–888. IEEE, 2009.

[9] M. Perez and T. Marwala. Stochastic optimization approaches for solving sudoku. *arXiv preprint arXiv:0805.0697*, 2008.

[10] M. Plauth, F. Feinbube, F. Schlegel, and A. Polze. A performance evaluation of dynamic parallelism for fine-grained, irregular workloads. *International Journal of Networking and Computing*, 6(2):212–229, 2016.

[11] K. Rocki and R. Suda. Parallel minimax tree searching on gpu. In *International Conference on Parallel Processing and Applied Mathematics*, pages 449–456. Springer, 2009.

[12] H. Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27. Citeseer, 2005.