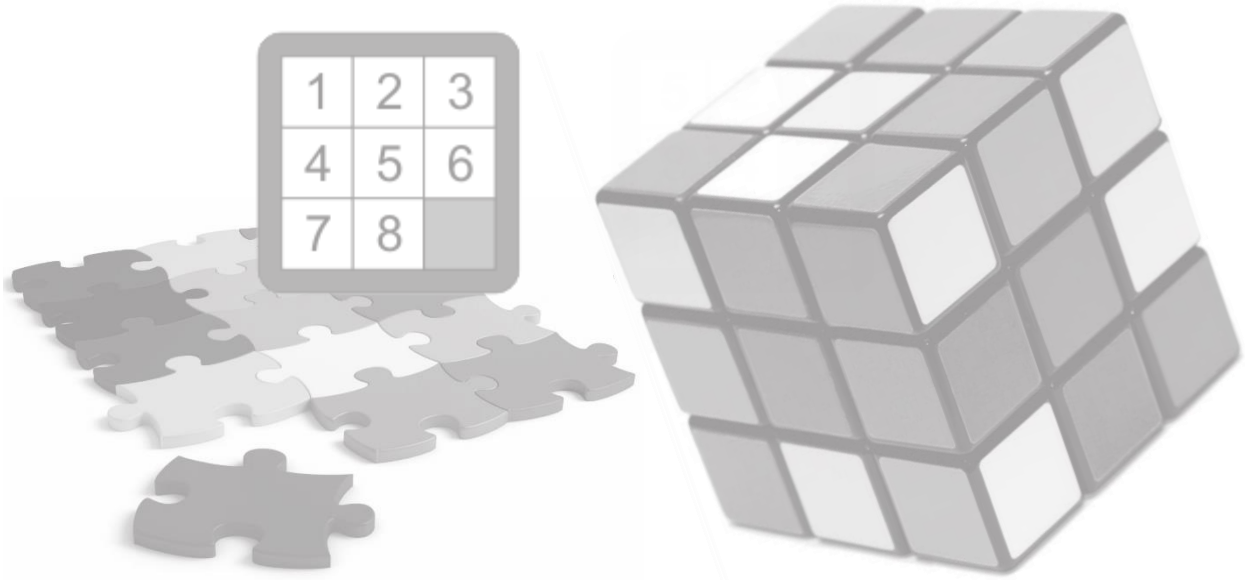
Chapter 4.

An Application of CUDA: *Puzzle Solving*



4. Introduction

Puzzles are fundamental things that we came to know since our childhood. For instance, we know Sliding Puzzles, Rubik Cube, Jigsaw Puzzle and many more. They are the real life entities of complex mathematical problems. Our objective is to solve a puzzle with minimum overhead/effort/moves i.e. reaching the goal faster. In real life, we face obstacles and challenges, which can also be mapped as solving a puzzle. Thus developing efficient algorithms for solving a particular puzzle has been a keen interest of the researchers.

4.1. Discrete optimization problem

A discrete optimization problem (DOP) is a class of computationally expensive problems which can be expressed as a tuple (*S*, *f*). The set *S* is finite or countably finite set of all solutions that satisfy specified constraints. This set is called set of feasible solutions. The function *f* is the cost function that maps each element in *S* onto the set of real numbers *R*.

$$f : S \rightarrow R$$

The objective of a DOP is to find a feasible solution x_{opt}, such that $f(x_{opt}) \leq f(x)$ for all $x \in S$ [.

Problems from various domains including puzzle solving can be formulated as DOPs. Some examples are test pattern generation for digital circuits, the optimal layout of VLSI chips, robot motion planning etc.

DOP has significant theoretical and practical interest. Search algorithms can be used to solve DOPs. Search algorithms solve DOPs by evaluating candidate solutions from a finite or countably finite set of possible solutions to find one that satisfies a problem-specific criterion. DOPs are also referred to as **combinatorial problems**.

4.2. The 8-puzzle problem

The 8-puzzle problem is the minor variant of the generalized n-puzzle problems. It consists of a 3 x 3 grid containing eight tiles, numbered one through eight. One of the grid segment (called “blank”) is empty. A tile can be moved into the blank position from position adjacent to it, thus careating a blank in the tile’s previous position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right as shown in figure 4.1. The initial and final configuration of the tiles are specified. The objectiveis to determnine aa shortest sequence of moves that transforms the initial configuration to final configuraton. Figure 3.2 illustrates simple initial and final configuration and a sequence of moves leading from the initail configuration to final configuration.

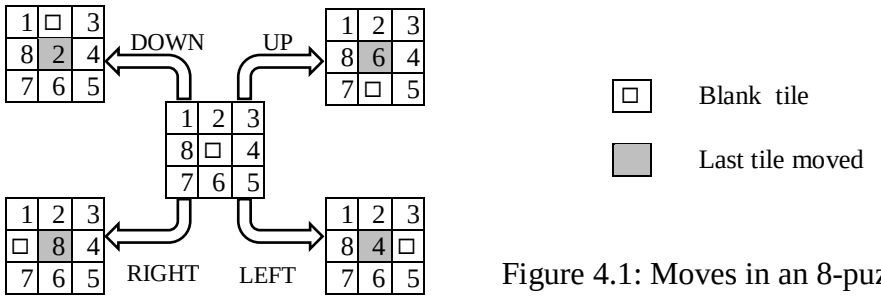


Figure 4.1: Moves in an 8-puzzle

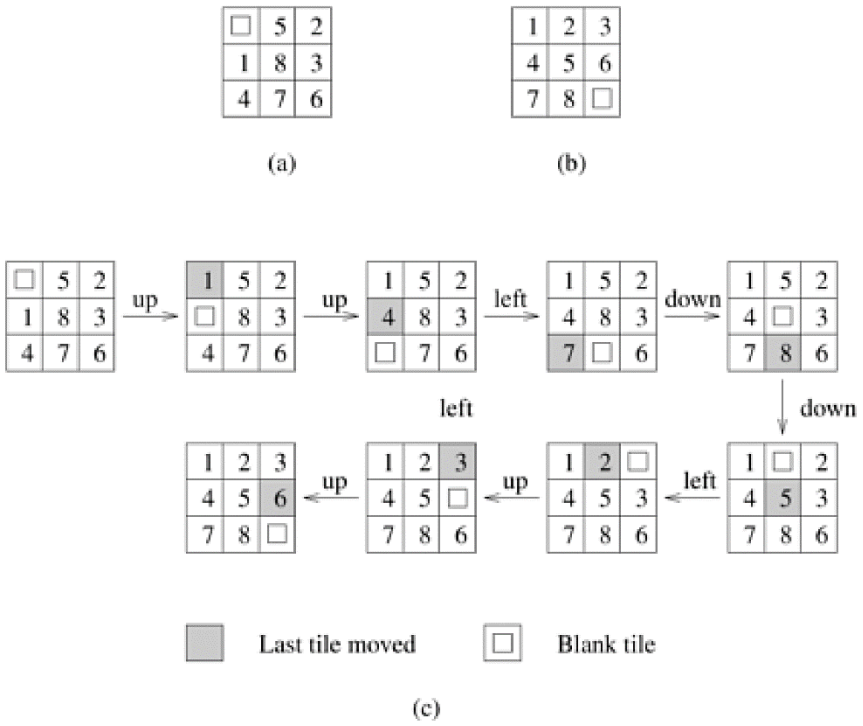


Figure 4.2: An 8-puzzle problem instance: (a) initial configuration; (b) final configuration; and (c) a sequence of moves leading from the initial to the final configuration

The set *S* for this problem is the set of all sequences of moves that lead from initaila configuration to the final configuration. The cost funtion *f* of an element in *S* is defined as the number of moves in the sequence.

In most problems of practical interest, the solution set *S* is quite large. Consequently, it is not feasible to exhaustively enumerate the elements in *S* to determine the optimal element x_{opt}. Instead, a DOP can be reformulated as the problem of finding a minimum-cost path in a graph from a designated initial node to one of several possible goal nodes. Each element *x* in *S* can be viewed as a path from the initial node to one of the goal nodes. There is a cost associated with each edge of the graph, and a cost function *f* is defined in terms of these edge costs. For many problems, the cost of a path is the sum of the edge costs. Such a graph is called a **state space**, and the nodes of the graph are called states. A terminal node is one that has no successors. All other nodes are called nonterminal nodes.

The 8-puzzle problem can be naturally formulated as a graph search problem. In particular, the initial configuration is the initial node or start vertex, and the final configuration is the goal node or terminal vertex.

As the 8-puzzle is reduced to a simple graph search problem, we can now take any graph traversal algorithm, like DFS or BFS, in order to find a solution path. Sincle our interest here is to build a prallel algorithm that finds the desired solution, we simply need to parallellize the search algorithms. For a long time it was taken for granted that DFS is inherently sequential [1]. Later parallel formulations for DFS have been proposed, most of which uses a load balancing scheme [2]. Since DFS is hard to parallelize we can use BFS as our underlying seaching method. Another reason for choosing BFS is that it gives the shortest path in case of unweighted graphs [3]. Thus BFS is well suited for this scenario.

4.3. Parallel Algorithm

Here we attempted to design an parallel algorithm which solves a given 8-puzzle instance. We used the parallel BFS algorithm discussed in section 3.2, with some minor modifications. First we need to set the initaial and goal nodes. Also we need to add an predicate which terminates the algorithm as soon as goal node is reached. Moreover we need some mechanism to obtain the path, which corresponds to the minimum number of moves. The outline of our algorithm is given as follows.

Algorithm: parallel_8_Puzzle_solver

1. Mark *initial* state as visited //a state corresponds to a vertex
2. Put initial on the *frontier*
3. For all states *v* on *frontier* do steps 4 to 8 in parallel
4. Remove *v* from *frontier*
5. If *v* is *goal* state then terminate
6. Compute next states of *v* by making valid moves
7. Put each new state on to *updating_frontier* if it is not already visited
8. Mark these states as visited
9. Synchronize all threads
10. Copy states from *updating_frontier* to *frontier* in parallel
11. Go to step 3

4.3.1. Analysis

To conduct our experiments we took Machine2, specified in section 3.5.2. The parallel algorithm was implemented using CUDA C/C++. We also implemented a sequential algorithm which was executed on the CPU of Machine2.

We have taken the goal state as shown in figure 4.2(b). We ran for various initial states and their solving times were recorded both on GPU and CPU. The Table 4.1 gives running time for few of the initial states. GPU kernel launch requires some extra computation like setting up CUDA context etc. This incurs extra overheads, which is clearly seen in row 1 of the table. For small number of moves parallel overhead is significantly high. But as the number of moves increase, i.e. workload increases GPU out performs the CPU as shown in 3rd row of the table. For higher order puzzles the gain is more likely to be very high.

Table 4.1

Initial State	Moves required	Sequential (Athlon II x4 630)	Machine2 (GT 730)
123456708	1	5 msec	121 msec
012345678	22	246 msec	288 msec
017254368	30	514 msec	402 msec

4.4. Finding Complete Solution

Another issue concerning with puzzles is to find complete solution. As the term suggests, complete solution is the collection of solutions of all possible permutation of the puzzle (solvable permutations only). This provides information like how many solvable states are there for a given puzzle. Which one is the hardest (maximum moves) puzzle or which one is the easiest. Thus, a complete statistical database is produces.

4.4.1. Procedure

Similar database can also be produced for 8-puzzle problem. A naïve way to obtain the database is to loop through all possible permutations and solve for each one of them sequentially. As one might expect this will take forever, some parallelism must be explored. As we have seen earlier, that 8-puzzle problem does not always efficiently utilize the total number of threads deployed; here we will solve 9! permutations with 9! threads each solving sequentially on its own. Hence, the algorithm can be outlined as follows:

Algorithm: complete_8_Puzzle_solver

1. Generate all possible permutaions of 012345678
2. For each permutaion *p* do steps 3 and 4 in parllel
3. Solve *p* sequentially with BFS
4. Store result

4.4.2. Optimizations

The sequential BFS procedure maintains a *queue* also, it needs some arrays for instance, an array to mark the visited states, an array to store the distances (number of moves) of each states. It may also have an array for keeping track of the path. This would requires huge amount of space when large numbers of threads run. Since GPU memory is significantly small compared to modern day RAMs, space requirements must be minimized.

Now some problem specific optimizations can be done here to reduce space requirements. First of all the *path* array can also be used as the visited *flag* array. For this initialize the path array with -1, which denote not-visited, and as the algorithm proceeds the *path* array is filled with state indices, which are non-negative values. Therefore, a non-negative value signifies the state is already visited.

Next optimization is bit trickier one. By logically thinking, one can say that only *distance* value of *goal* state is needed. There is no need to keep distances for every other states. Therefore, a single variable might do the job. But, the *distance* array is used to calculate distances of explored neighbours by the following formula: say from state *x* we visit state *y*, the *distance[y] = distance[x] + 1*, starting with *distance[source]=0* initially. Utility of distance array can be purged by keeping a flag value in the BFS queue. How it actually works is explained inside the algorithm. Hence all we need a single *path* array, of size 9!, instead of three arrays. The outline of the algorithms is given as follows:

Algorithm: complete_8_Puzzle_solver

1. Add *source* state into the *queue* and set its parent to a dummy value, say zero, into *path* array.
2. Also, add a flag value, say *-1* to the *queue*.
3. Initialize *count* as 0
4. While the *queue* is not empty do steps 5 to 12
5. Take one item out of the *queue* and store it in *cur*
6. If *cur* is *goal* then update *cost* with *count*, and break out of the iteration
   //early termination of BFS
7. If *cur* is not *-1* then do steps 8 to 11 //an actual state to be processed
8. Generate a *next* state by making a move on current state
9. If *next* is not a valid state continue with following move
   //result of attempting to make a move which is not possible
10. If it was a valid move, obtain index of next state, set *path[index]* = index of current state and put *next* state in the *queue*. These are only performed if the next state is not already visited //*path[index]* was set to *-1*.
11. Perform steps 8 to 11 for all four moves
12. If the *cur* is *-1* then we encountered a flag value, which signifies two things. First, all nodes enqueued before it, are dequeued and processed. This resembles end of a level, so increment the *count* by one. Second, all nodes of next level are already in the queue but yet to be processed. So, it is now time to again enqueue a *-1* as flag to the next level. Before putting the *-1* into the queue check if the *queue* is not empty, otherwise it will lead to an infinite loop.

4.4.3. Analysis

To conduct our experiments we took both Machine1 and Machine2, specified in section 3.2.4. The optimized algorithm was implemented using CUDA C/C++. We also implemented a sequential algorithm which was executed on the CPU of Machine2.

We took the goal state as shown in figure 4.2(b). For 181440 permutations, which is half of total 9! permutations, there were no path to reach the goal state, i.e. only half of total possible solutions are solvable. This supports the theoretical claim that half of the starting positions for the n-puzzle are impossible to resolve, no matter how many moves are made [4].

For the permutation, which is the goal state itself, zero moves were required. For those permutations which are not in goal state, minimum and maximum moves required were 1 and 31 respectively. Highest numbers of permutations (24047) are solved in 25 moves, while 23952 permutations are solved in 23 moves leading to the second highest. There are only two permutations solvable in 31 moves. The distribution of solvable permutations over required minimum number of moves to reach the goal node is shown in the graph (Figure 4.3).

For limited available space on the GPU, we ran the parallel algorithm to solve a batch of 504 states. Thus, total 720 such batches were executed. This distribution is for Machine1. However, for Machine2 we ran batches of 1008 states, requiring 360 such batches. The time taken by the algorithms are given

in Table 4.2. The results clearly demonstrates the power of CUDA threads. It also suggests that as the underlying GPU technologies advances it becomes more capable and powerful.

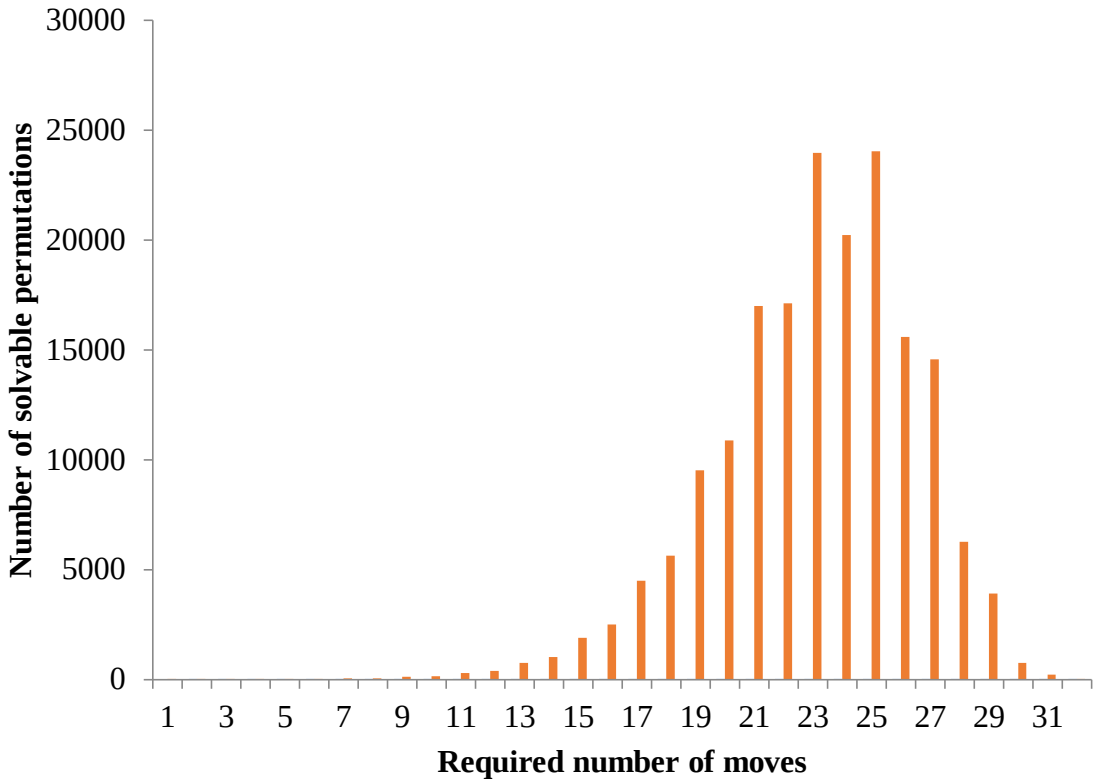


Figure 4.3: Distribution of solvable permutations

Table 4.2

Sequential (Athlon II x4 630)	Machine1 (GT 520)	Machine2 (GT 730)
A little over 20 hours	13585 seconds	5369 seconds

References

[1] John H. Reif, "Depth-first search is inherently sequential" at Center for Research in Computing Technology, Aiken Computation Laboratory, Division of Applied Science. Harvard University, Cambridge, MA 02138, U.S.A

[2] A. Grama, G. Karypis , V. Kumar , A. Gupta, "Introduction to Parallel Computing"- 2nd edition

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms"

[4] W.W. Johnson and W.E. Storey. "Notes on the `15'-Puzzle." Amer. J. Math. 2(1879), 397-404