Finding solutions to the n-queens puzzle on GPU

Anders Asheim Hennum

EPFL, June 8th

Abstract

This paper is about using GPU to find solutions to the n-queens puzzle. The puzzle was solved by exploring the solution tree with random first search. This is a general constraint satisfaction algorithm and can be applied to large number of problems. Compared to a CPU performing the same number of searches sequentially, the GPU reduces the execution time ~ 7.

1 Introduction and description

Solving problems by searching for solutions is a commonly used method in artificial intelligence. It is a general way of solving problems and, thus, has good capabilities to solve a large number of unrelated problems. In this paper I have chosen to work with the n-queens puzzle [2]. It is a simple and well documented puzzle where the goal is to place n queens on an $n \times n$ chess board without any queen attacking another queen. There are several ways to solve this puzzle. Here, a variant of constraint satisfaction [3] has been implemented. It can intuitively be viewed as as a random first search where the the tree of all partly and full solutions is explored until a solution is found. This approach relies on random searches and has very variable running times. By using GPU, we can run multiple searches in parallel and by then, have a greater chance of finding solutions faster than running multiple searches on CPU sequentially. In this paper the same algorithm for solving the puzzle has been implemented on CPU and GPU and performance has been compared. All source code and benchmarking code and results is available on Github [1].

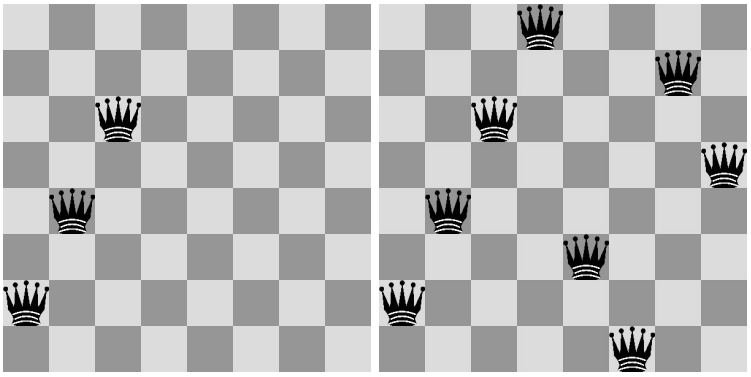
2 Implementation

2.1 The Algorithm

In short words the algorithm works by placing out a queen randomly in the first column. It continues with placing a queen in the next column and makes sure that the row and diagonal is free and that the position has not been tried before. When a position is found where all constraints are met (row and diagonal

is free) it continues to the next column. It does this for all columns (levels in the tree) until it reaches a dead end or finishes by successfully placing a queen in the last column (reaches the bottom in the tree). If a dead end is reached (no next placements possible) it back tracks to a previous column where not all possibilities is tried yet and continues to search from there. Pseudo code of the implementation of the algorithm is in the appendix. It has been implemented with a limited number of iterations since a search quite often get stuck somewhere in the solution tree far from a solution and will use significantly longer time to find a solution than starting over again.

For a board of size $n = 8$ there is 4,426,165,368 possible arrangements of eight queens and only 92 solutions [2]. The algorithm deals with this by reducing the search space. It considers only arrangements that are partly solutions. In figure (1) we see a partly solved puzzle and a final solution found by the algorithm.



(a) Partly solved puzzle (b) Full solution of puzzle

Figure 1: Solution progress of n-queens puzzle with $n = 8$.

2.2 Implementation on GPU

The algorithm can more or less be implemented directly on the GPU, but some things should be thought of i terms of efficiency. There is mainly three things we need to consider: random number generation, memory management and thread divergence. See source code on Github [1] for full details on implementation.

2.2.1 Random number generation

In this approach to solve the puzzle every thread needs to generate a large amount of random numbers. To obtain good performance this must be done in an efficient way. The CUDA toolkit has a good library `curand` [4] for this. The documentation describes how to obtain highest quality parallel pseudorandom number generation. I general, every experiment is assigned with an unique seed value (time). Within the experiment all threads is assigned with an unique id

number (by thread id and block id) and this is used as sequence number. By this we are guaranteed that all threads will get different sequences and thus will perform different searches. This is the recommended way to generate pseudo random numbers with CUDA and has been implemented here. See source code for details.

2.2.2 Memory management

The algorithm is heavily dependent on reads and writes from memory. Each thread must maintain it's current (partly) solution, an array of the domain (free rows) and a 2D array of positions tried at each column. The two options available is shared memory and local memory (which practically is global memory). Shared memory is much faster than global memory and shared between threads in the same block. Shared memory can be accessed in parallel where as global memory is accessed sequentially. To get maximum performance one should maximize the use of shared memory to what is possible. An attempt to use shared memory was done here, but due to some issues and limited time this did not work out and global memory was used. This is definitely the bottleneck and for further work making use of shared memory should be considered.

2.2.3 Thread divergence

On the GPU all threads within the same warp executes the same instructions. To avoid too much branching that will result in threads out of sync and performance loss, the algorithm has been implemented in an iterative way instead of recursive. In this way all threads will execute the same loop all the time and stay in sync. Thread divergence is then minimized and better performance is obtained.

3 Results and discussion

Benchmarking was done on a Nvidia Quadro K2000 GPU card and a 2.5 GHz CPU. The results is based on averages of five runs. Figure (2) shows execution time as a function of searches performed. We can observe that we have a linear relationship between execution time and number of searches. The speedup seems to be more or less constant. Table (1) shows results for various setups. The GPU and the CPU finds on average about the same amount of solutions indicating that their random generators perform equally good. Speedup is measured from 5 to 7 with a peak at 7.36. Varying grid size does not seem to affect the speed up notably in this problem. However, we do have a maximum speedup at a grid size of 16 blocks and 128 threads for both problem sizes. This indicates that this grid size is optimal for this problem on the specified hardware. We also see that we have higher speedup at a problem size of $n = 40$ than $n = 100$. This is probably due to the memory access pattern of the GPU. It suffers more with high memory latency on larger problems.

Table 1: Benchmarking results. Values are averages of five runs.

n	Searches	Blocks	Threads/block	Time GPU [ms]	Time CPU [ms]	Solutions GPU	Solutions CPU	SpeedUp
40	256	16	16	40	134	18	16	3.35
40	512	16	32	52	264	30	29	5.08
40	1024	16	64	82	518	93	65	6.32
40	2048	16	128	144	1060	111	124	7.36
40	4096	16	256	298	2034	227	253	6.83
40	8192	16	512	612	4096	490	497	6.69
40	16384	16	1024	1230	8028	1029	1008	6.53
40	1024	32	32	78	508	52	57	6.51
40	2048	32	64	146	1002	130	136	6.86
40	4096	32	128	304	2008	251	247	6.61
40	8192	32	256	610	4014	513	501	6.58
40	16384	32	512	1228	8008	1010	1001	6.52
40	32768	32	1024	2442	16044	1891	2051	6.57
40	4096	64	64	290	2008	251	247	6.92
40	8192	64	128	616	4004	518	514	6.50
40	16384	64	256	1222	8032	983	992	6.57
40	32768	64	512	2438	16028	2051	2030	6.57
100	1024	16	64	234	1228	4	2	5.25
100	2048	16	128	440	2442	7	7	5.55
100	4096	16	256	912	4904	19	11	5.38
100	8192	16	512	1840	9682	33	25	5.26
100	16384	16	1024	3674	19238	52	49	5.24
100	4096	64	64	870	4806	14	14	5.52
100	8192	64	128	1834	9626	23	22	5.25
100	16384	64	256	3660	19238	48	49	5.26

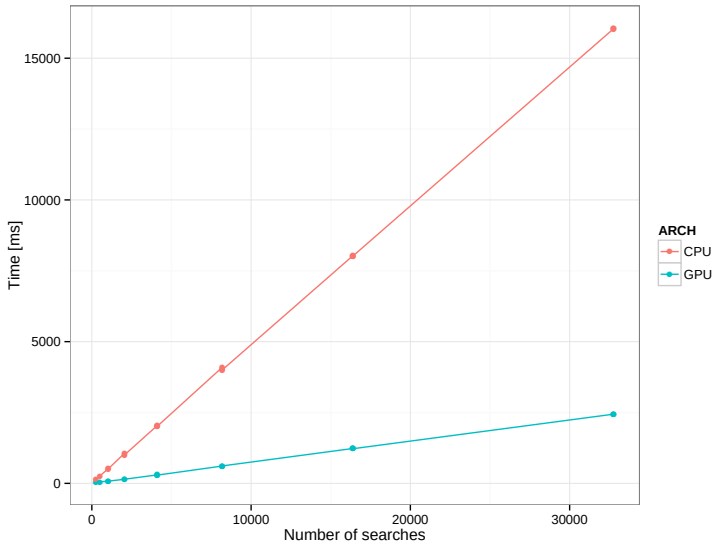


Figure 2: Number of searches performed vs. execution time with a board size $n = 40$.

4 Conclusion

We have seen that it is possible to solve the n-queens puzzle on GPU. Even with not optimal use of it's computing capabilities it still reduces the execution time by ~ 7 compared to a single CPU. By making use of shared memory on the GPU I believe this can be reduced further. It should be a topic for further work.

Not much work has been done on using GPU for these kind of problems. With the large amount of problems that can be solved by this approach I think it is potential for further research.

References

[1] `http://github.com/hennumjr/nqueengpu`

[2] `http://en.wikipedia.org/wiki/Eight_queens_puzzle`

[3] `http://en.wikipedia.org/wiki/Constraint_satisfaction_problem`

[4] `http://docs.nvidia.com/cuda/curand`

A Appendix

Algorithm 1 Solution search for n-queens puzzle

```
1: initialize n, S, N, D, i, q, iter, max_iter
2: while ( iter < max_iter ) do
3:     q ← rand()*n
4:     if (D[q] == 0 and N[i][q] == 0) then
5:         N[i][q] = 1
6:         if (diagonalsOn(q,i,S) == 1) then
7:             S[i] = q
8:             D[q] = 1
9:             i++
10:            if (i == n) then
11:                brean
12:            end if
13:        end if
14:    end if
15:    if ((sum(N[i]) + sum(D)) == n) then
16:        D[S[i-1]] = 0
17:        S[i-1] = -1
18:        for each j in N[i] do
19:            j = 0
20:        end for
21:        i–
22:    end if
23:    iter++
24: end while
```

Variable description:

- n: (Int) Board size
- S: (Int[n]) 1D array where S[i] = j means that there is a queen at position i,j.
- N: (Int[n][n]) 2D array that holds the positions tried at each column. Values are 0 (not tried) and 1 (tried).
- D: (Int[n]) D[i] == 0 means that row i is free, 1 means tanen.
- i: (Int) current column
- q: (Int) Random number. The current position being tried.
- iter: (Int) Iteration counter
- max_iter: (Int) Iteration limit.

Line comments:

4: Checn if row is free and position not tried before.

5: Marn position as "tried" i.e change from 0 to 1.

6: Checn if diagonals are free and proceed.

7: Add new queen q to S, the solution array.

8: Set row as "used" i.e change from 0 to 1 in D[q].

9: Increment column identifier.

10: Checn if it is finished.

15: If this sum is equal to n then all positions is tried and no new placement is possible. A dead end is reached.

16: Set row of last queen as free again

17: Remove queen from solution, i.e set to -1

18,19: Reinitialize positions tried for current column.

21: Decrement column identifier. Bacntracning.

23: Increment iteration counter.