N-Queens Solution Implementation on NVIDIA GPU using CUDA

Conner Wulf, Derek Rodriguez, David Hoambrecker

Abstract: This paper explores the N-Queens problem and how we implemented it on a GPU using CUDA, and optimized it using shared memory.

I. Introduction

The N-Queens problem asks to solve the number of solutions in which N number of queens can be placed on an NxN chessboard without attacking each other. This means that no queen can be in the same row, column, or diagonal as any other queen on the board. Specifically the 8 Queens problem was proposed by chess composer Max Bezzel in the mid 19th century. Shortly thereafter, Franz Nauck published the first solutions and extended the original problem to the N-Queens problem for any board size.

There exists solutions for any integer board size except for the values n = 2 and n = 3 due to the limited size of the board. The number of solutions per board size rises exponentially as N increases, which typically calls for a recursive backtracking solution when using a CPU implementation. This problem was first used to demonstrate the power of depth-first backtracking in computing by Edsger Dijkstra in 1972. This backtracking method is accepted as the most efficient way to solve the n-queens problem, but even with the best hardware available the runtimes spike dramatically for values of n > ~14. The highest board size for which solutions have been calculated is n = 26; beyond this point the number of solutions has never been found. Given the well-researched and published history of the N-Queens problem, we felt that a parallelized GPU solution could offer an interesting insight into how GPU performance compares to CPU performance for the same problem. Unfortunately, recursion and CUDA do not play well together, so a different approach was needed for our solution.

[1][1]https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

II. The Code

The first step was to produce a working N-Queens solution that can run on a CPU. This was done by representing the board of queens as a 2D vector which scales to the appropriate input size N. These boards are recursively generated and checked until each possible solution is accounted for.

We encountered a big issue with our initial idea and algorithm design. We implemented the original CPU code via a recursive backtracking algorithm. We had to come up with a clever solution to implement the code for parallelization. We discovered that each solution could be represented by a tuple of size N, which would help ensure that the number of threads in each block was N × N. We assigned values to tuples based on the x and y block IDs and thread IDs, which allowed us to access these in constant time. We were able to consider every possible permutation through iterative loops instead of recursion.

Only half of the solutions of the board needed to be generated due to symmetry of the boards. This means we calculate all possible solutions by storing whether or not a solution was present in an array stored in shared memory, summing up the number of of total solutions, and doubling them for the total number of possible solutions for any board of N × N size. To find these solutions, we check every possible direction for an attacking queen. If no attacking queen is present, we add a 1 to the array of solutions. This is demonstrated in Figures 1 and 2.

```
for(int k = NUM - 1; k > wrongCount * NUM; k--){
  for(int m = k - 1, count = 1; m >= 0; m--, count++){
    wrongCount += tups[threadIdx.x][threadIdx.y][k] == tups[threadIdx.x]
    [threadIdx.y][m];
    wrongCount += (tups[threadIdx.x][threadIdx.y][k] - count) == tups
    [threadIdx.x][threadIdx.y][m];
    wrongCount += (tups[threadIdx.x][threadIdx.y][k] + count) == tups
    [threadIdx.x][threadIdx.y][m];
  }
}
sol[threadIdx.x][threadIdx.y] += !(wrongCount);
wrongCount = 0;
```

Figure 1: Example code checking directions of attacking queens

```
for(int t = 0; t < num_Blocks; t++)
{
  *d_count += d_answer[t];
}
*d_count *= 2;
```

Figure 2: Example code summing up the number of solutions.

III. Conclusion

The results for this project were not what we expected. Going into this class we knew we wanted a project that would show how superior GPU programming can be to the generic CPU programming. Our GPU version ended up being slower that our CPU code, which was disappointing. During the development of our project we began to realize that the exploration of the N-Queens problem on a GPU had not been heavily explored and there is no general consensus as to whether it is a parallelizable problem or not.

Our program falls on the side of the CPU program being the more optimal choice. We discovered that regardless of how we tried to optimize the program, the CPU would, on average, run around $\frac{1}{10}$ th of a second faster than the GPU at our best optimizations. Although our program executed in parallel, it wasn't able to fully utilize the maximum number of cores and threads available via the GPU. We essentially utilized one thread per one board position, which is very useful and efficient, but for a problem of this nature wasn't sufficient. We were unable to efficiently solve this problem through parallel programming, and we did further research upon discovering this.

There is much debate among the scholarly community as to whether or not this problem can be efficiently solved through parallel programming on the graphics processing unit, and we found research exclusively stating why this isn't an optimal problem for parallel programming.

As a group, we learned that with the modern developments of GPU hardware, parallel processing is a promising field of study that can yield extreme performance gains over typical CPU programming. However, we also learned that to benefit from this processing power, the problems must be absolutely parallelizable, and we learned that not every problem could benefit from parallel processing power. On top of that we also realized how crucial it is to make sure that programs are optimized correctly, using as much GPU power as the problem allows. This can come in the form of maximizing thread usage, optimization techniques, and solid algorithm design. We hope to be able to continue learning about CUDA throughout our professional carrers.