Jonathan Lehman

Homework 5
N Queens Problem

The N queens problem finds the number of solutions in which N queens can be placed on an N x N board without attacking each other, that is the queens cannot be in the same row, column, or diagonal as any other queen on the board. Typically this is done with recursion and backtracking, but since recursion will not work with CUDA, another approach must be determined.

Initially, I planned on checking each possible board combination and evenly dividing the work up between the threads available. Using shared memory, each thread would sum up the number of correct solutions out of the combinations it checks. Then each block would sum up these solutions to get the number of solutions per block, which would be returned in array to be summed up by the CPU. I also realized that there were mirror image solutions that could be ignored, but I was not positive that this was true for half the board. The problem was, I did not know how to best represent these combinations. I thought the best way to represent the placement of each queen would be with a pair of coordinates.

In class we discussed a potential solution to the N queens problem where each solution could be represented by a tuple of length N, where each value in the tuple can be 0 to N – 1, meaning the modulus becomes a very useful operation in the solution. We could also ensure that the threads in each block are N x N. By doing this we can assign values to tuples based on blockIdx.x, blockIdx.y, and the thread IDs. Each of these 4 values can be assigned to a tuple value in constant time. Any of the remaining values (N – 4 values) in the tuple can be generated iteratively for each thread to account for all possible permutations using for loops. In addition, since the first value of the tuple (generated by blockIdx.y) is based on the row number of the block where the value is between 0 and N – 1, and as I mentioned before, only half of the possible tuple solutions need to be examined, because the other half of the board will just produce reflections of the same solution, therefore the number of rows must only be 0 to N / 2 + (N%2), which essentially is the ceiling of half of N. Then during the summation of the solutions process, the number of N/2 rows can be summed then doubled, and if N is odd the additional row sum can be added as well.

Now I will go through what my code does, and then explain any major alterations I made to increase the efficiency, of which I made several. To begin with the macro _N_ is used to define the board size as well as the number of queens to be placed on the board. The macro sumOnGPU, which is 0 by default, lets the user specify whether the block sums should be totaled on the GPU or CPU (0 for CPU and 1 for GPU). The macro numBX, defines the number of tuples that are generated by blockIdx.x. For example, if this value was two, the width of blocks in the grid would be N^numBX, and the tuples would be calculated with a modulus. At the beginning of my code, I check to make sure that all macros are within their allowable range. For example, numBX cannot make N^numBX larger than the maximum grid width of about 65000. Also, N must be between 4 and 22 inclusive, 4 because there are always at least 4 tuples generated by the block and thread IDs, and 22 because that is the maximum dimension of the number of threads in a square block without allowing the number of threads to go over the maximum block size of 512 threads. The grid width is than calculated by the numBX macro, and an array "a" is allocated such that there is an element for each block. This array will hold the blocks sums, which will either be passed back to the CPU to be summed, or summed in a device method on the GPU where the sum is stored to the first value of the array and returned to the CPU. Then the method I used in my last project, checkGPUCapabilities, checks to ensure that the grid and block sizes, and that the memory of the array does not exceed the maximum global memory on the GPU. This uses queries so it will work on any GPU. The memory for the array "a" is than allocated on the GPU, using CUDA events, the timer begins and the CUDA kernel is called. I will explain the logic of this kernel once I finish explaining the CPU code. Once the GPU finishes execution, the CUDA event is used to stop the time, and the memory is copied back to the CPU, which would be the array "a". If any errors occurred on the GPU, the last error to occur is printed on the screen, otherwise it prints that no errors occurred on the GPU. Then if the sumOnGPU macro specified the program to sum on the CPU, the CPU timing mechanism used on the sumseq project is used to record the start time to do this work on the CPU. Each of the block totals (element in "a") is summed, and doubled (except the last row if N is odd), then the finish time is recorded and used to find the total summation time on the CPU. The program then prints the number of solutions, which would be the sum just calculated, or if the sum was done on the GPU, it is located in the first element of "a", as well as the time, or times if the CPU was used to sum.

The initial kernel first declared two arrays of shared memory, one for storing the each thread solution in the block, and the other for each threads tuple. The second array was not really "shared" but using shared memory to access the tuple provides faster times. The first step was to generate the tuple. The first value was set to the blockIdx.y value, or the block row number (this is the tuple that has been halved from N values to N/2 values to account for the mirror image solutions that are taken into account during the summation process). The next numBX values were set using the blockIdx.x value, or the block column number. To generate values when numBX > 1 meaning the number of columns > N, the program relied on a device method that convered a number in base 10 to base N, and stored each value in an array of length numBX, with padded 0's at the front if necessary. Each element of this array was used as a tuple value. The base 10 number used was the blockIdx.x value. Next 2 tuples were set, one to threadIdx.x and the other to threadIdx.y, the column and row number of the current thread working. These tuples remain the same for the thread, while the thread generates the remaining numGen (a macro not specified by the user, containing the remaining number of tuples to generate) tuples, checking if each full tuple set is a solution. Before the iteratively generated tuple values are dealt with, the thread checks to see if the constant values are valid, before doing any work that may be unnecessary, for example if there are already two tuples with the same value, meaning the queen is in the same row. Also, it was not mentioned earlier, but the column does not need to be checked, because the tuple representation does not allow any queen to be in the same column, therefore only the row and diagonals must be checked. To check these beginning tuple values we iterate backwards through the tuples, going from index N-1 to 1, with a for loop, and have a nested for loop going backwards from the current value of the first for loop to 0. Inside the nested for loop three checks are done. If the values at the indices in the tuple are equal, meaning they are in the same row, or if the value of the index of the outer counter is equal to the value of the index +1 or -1, the upper left lower left diagonal. Rather than using if statements, which are less efficient in CUDA because the threads split and the warp does not stay together, a counter totalWrong is incremented by the value of the conditional expression. Therefore, for each error, totalWrong is incremented by one (true) and incremented by nothing on false, such that it can remain 0 if nothing is wrong. If there is totalWrong is not 0, none of the remaining tuples, if any, are generated because the tuple would not be a solution regardless.

If totalWrong is still 0, a for loop from 0 to N^numGen, iterates and the method to convert the base 10 number (the loop value) to a base N array and store these values to tuples is utilized. In other words for each thread it tries all of the possible tuple values for the tuples not yet filled in, and checks each full solution using the for loop decrementing through the tuple as described above. If totalWrong is 0 at the end of the loop that checks for correctness for each tuple, it is the number of solutions for that thread (stored in the shared array for solutions for the block mentioned at the beginning of the kernel section) is incremented by 1. Rather than using an if statement again, it simply increments by !(totalWrong), 0 if the solution is not valid, and 1 if it is. After this process the threads are synchronized, using __syncthreads, to ensure that each has found the number of solutions it is responsible for. Then thread 0, the first thread in each block, sums up the solutions for each thread by iterating through the shared array for the block, and stores the value into the global memory array "a" at the index corresponding with that particular block number. Then if sumOnGPU is set to 1 the first thread in the first block goes to a device method and sums all of the values in the array "a" to get the total number of solutions (doubling where necessary), which is the exact same summation process used on the CPU as described in above, except that the resulting value is stored in the first element of "a". This was my initial solution, and all future amendments to this solution described in this report will remain exactly the same except for the explicit changes described. Also each modification will be built upon, such that the first modifications will remain unless explicitly stated otherwise.

First of all, the problem with this approach was that I used additional global memory on the device, which not only used more memory, but had slower access time by my use of the method that converted base 10 numbers to the base N number in the form of an array padded with 0's at the front as necessary. This method was called often, and could vitally slow down my timings because of the use of global memory, and it being fairly inefficient overall. I replaced this method with a for loop that takes modulus N of the initial base 10 value, which becomes the first tuple value, and makes the new base 10 value the old value divided by N. This for loop repeats until the number of tuple values to be filled have been filled, which may result in some of them being 0, the same effect that padding had. This provided the same effect, but did not use global memory, and was far more efficient. I also replaced some of the global memory accesses to the array "a" with a local variable to store the current sum in and only store that value to global memory after the full summation had been done to reduce the number of global memory accesses. These initial timings can be found in **Figure 1**.

My next modification was only a minor modification, but it greatly reduced timings. I realized that my method of generating the last tuple values to be filled in for each thread (if any) was very inefficient. It checked every possible value in base N. For example if there were 2 tuple values to generate it checked from 00 to NN, but both of these extremes are clearly not solutions because they have queens in the same row. The best method would be to generate each permutation, such that each tuple only has no more than one of each value, meaning that only diagonals would need to be checked. I had difficulty finding an algorithm to generate a permutation. It seemed easier on the CPU where recursion could be used, and the only solutions I was able to come up with seemed fairly inefficient. I even tried to pass all of the permutations I generated on the CPU in as an array to the GPU, but this used up a lot of the GPU memory that I needed for the rest of my algorithm. After being discouraged by this, I decided to take the next best step, and I added a for loop to check that the generated tuple values were not identical. This was much quicker than the for loop that checks the entire tuple, and prevents the longer for loop for executing for a large amount of the tuple values being generated iteratively. The timings from this change can be found in **Figure 2**. Overall there was a great decrease in the time. For example, when N=11 and numBX=1, which meant that 7 tuple values were being generated for each iteration by the thread, the first timing was 831.376375 seconds, while the modified solution's timing was 364.637563 seconds, which more than halved the initial time it took. All of the other timings that were done decreased as well (except in the case when no tuple values were being generated in this manner, numGen=0, where the timing stayed the same).

Despite this increase in efficiency, the runtime for any N over 11 still has a very long runtime. To try to further decrease the timings, I added another macro, numBY, which makes use of the grid's height, which currently is not being utilized. Essentially, it creates numBY rows, where the number of rows is a multiple of the original number. The work of generating the tuple values that do not get their values in constant time from a block or thread ID, is divided amongst these new blocks. This helps further parallelize the work being done, so more can be done at once. The only additional logic needed in the kernel is to identify which original row the row belongs to, for example if N = 6, there are initially 3 rows, so row 3 would be doing a share of row 0's work, row 4 would be doing a share of row 1's work etc. These values are easily determined using modulus arithmetic. Unfortunately, this modification did not reduce the timings as much as I had hoped (see **Figures 3 and 4**). It seems to peak at a particular point, it seems when each one is only handling a few thousand different tuples to check as potential solutions, but past that, performance deteriorates. For example, when N=11 and numBX=1, if numBY=1 it takes 346.812844 seconds to find all the solutions. If numBY is increased to 5000 the time decreases by 100 seconds to 347.414 seconds, but when numBY increases again, the performance does not get any better and stays at 247.574 seconds.

Though these optimizations greatly decreased the timings from my initial solution, there still are some areas that could be improved upon. There are a lot of wasted blocks and threads that do not do any work because they cannot possibly generate any solutions because the queens would be in the same row because two or more of the tuple values are the same (block id and thread id are the same etc). These could be avoided by using a permutation generation algorithm such that only possible solutions are generated (only diagonals would need to be checked). Also, in the same way that numBX greatly decreases the timings by completely removing tuple values that need to be generated iteratively (which takes a lot of time to generate all the possible combinations once there is more than one tuple value being generated), numBY could be used to generate multiple tuples based on the row number, however this would be much more complex because the first tuple number only needs to go to half of N rather than N-1 like the rest of the tuples because of the mirror image solutions that can be found just by doubling the solutions found on half the board.

To begin with, not all the times were found between 4 and 20. The problem with only one person being able to run something on the GPU at a time is that the GPU takes a while to time out, so it will not necessarily tell you that the code failed to execute for a few minutes, meanwhile you think it is solving the problem. I ran a trial with N=13 and numBX=4 (the maximum to decrease the time) and numBY=1000 (to further decrease the time), and after 54 minutes it still did not return a solution. Either it did not get on the GPU right away and did not take 54 minutes, or it would take longer than 54 minutes, but I decided to abort the test so that I wouldn't hog the GPU if someone else needed to run something. I was able to generate solutions up to N=12, and the code can handle up to 22 with enough memory and resources on the GPU, although this would probably take an extremely long time to solve. Lastly, each of the runs was on GPU1 or 2 on the lab machines while many other users were on the machines, and is an average of 3 runs.

It seems that it does not really matter whether a second kernel sums up the total or if the work is done on the host. The CPU summation times (**Figure 3**) are generally faster the than the GPU only times (**Figure 4**), but by a very small portion of the total time, so this number did not seem to be large enough to say that it was faster on the CPU, as they were about the same. The reason it was no faster on the GPU is because I used the same process to do the summation, so it was summed by one thread in serial, just like it was summed in serial on the CPU. So it makes sense that these times should be very close to one another. If done in parallel using a mutex (or similar functioning construct), it would probably be faster on the GPU to sum for large grid sizes.

On the question if more blocks should be assigned to the streaming processors or if each thread should examine more combinations of board configurations, has a very clear answer based on the data. It is much faster to use more blocks (increase the macro numBX in my solution) because than more work is done in parallel, rather than each thread doing more work in serial. To prove this point, when N=11 and numBX=1 (the block ID of the column only generates one tuple value) it takes 346.812844 seconds to solve (**Figure 3**), but when numBX is increased to 4 (the block ID of the column generates 5 tuple vales so less work is done by each thread), the time significantly decreases to 32.946035 seconds (**Figure 3**).

By only considering half of the possible combinations based on the placement of the queen in the first column, we save a considerable amount of time. It seems that one would not save much time because the other half that we are now considering is being done in parallel, however, there is a cost with using more of the GPU's resources, though not a lot, but enough to notice a difference in performance. It seems though that the biggest difference would be in the total time it takes to add up the blocks, since there are now far more blocks to add, but the difference is not only in the time of the sum being done by the CPU, but the work on the GPU as well. This probably has something to do with how warps and threads/ resources are handled on the GPU. This was tested on with the summation the CPU (**Figure 5**), so that the time of the summations could be compared as well as the run time on the GPU, and both increased. For example, when N=10 and numBX=4 and numBY=1, the time when only considering the half of choices the placement of the queen in the first column is 0.723600 seconds on the GPU and 0.000166 seconds on the CPU (**Figure 3**), and both of these values increase to 1.678810 seconds on the GPU and .000363 seconds on the CPU (**Figure 4**), which is a considerable increase in time. Based on these times it saves time to only consider half the possible combinations, because it does cost something to simply check all of the solutions on the GPU, even in parallel.

Figure 1: Initial Timings

N	numBX	numGen	GPU Time	CPU Time	Total Time	Solutions
4	1	0	0.000043	0.000001	0.000044	2
5	1	1	0.000061	0.000001	0.000062	10
5	2	0	0.000046	0.00001	0.000056	10
6	1	2	0.000217	0.000001	0.000218	4
6	2	1	0.000075	0.000002	0.000077	4
6	3	0	0.000096	0.000003	0.000099	4
7	1	3	0.002375	0.000002	0.002377	40
7	2	2	0.000402	0.000025	0.000427	40
7	3	1	0.000409	0.000157	0.000566	40
7	4	0	0.001353	0.001093	0.002446	40
8	1	4	0.039632	0.000001	0.039633	92
8	2	3	0.010471	0.000002	0.010473	92
8	3	2	0.004411	0.000021	0.004432	92
8	4	1	0.005088	0.000061	0.005149	92
8	5	0	0.033687	0.000471	0.034158	92
9	1	5	0.931559	0.000002	0.931561	352
9	2	4	0.394047	0.000049	0.394096	352
9	3	3	0.140401	0.000438	0.140839	352
9	4	2	0.061189	0.003824	0.065013	352
9	5	1	0.085937	0.033839	0.119776	352
10	1	6	21.254831	0.000001	21.254832	724
10	2	5	8.740024	0.000003	8.740027	724
10	3	4	3.987558	0.000018	3.987576	724
10	4	3	1.498701	0.000167	1.498868	724
11	1	7	831.376375	0.000001	831.376376	2680
11	2	6	338.480844	0.000088	338.480932	2680
11	3	5	189.426594	0.000981	189.427575	2680
11	4	4	71.830187	0.010587	71.840774	2680

Figure 2: First Optimization

N	numBX	numGen	GPU Time	CPU Time	Total Time	Solutions
4	1	0	0.000043	0.000001	0.000044	2
5	1	1	0.000058	0.000001	0.000059	10
5	2	0	0.000046	0.00001	0.000056	10
6	1	2	0.000195	0.000001	0.000196	4
6	2	1	0.000067	0.000002	0.000069	4
6	3	0	0.000096	0.000003	0.000099	4
7	1	3	0.002024	0.000002	0.002026	40
7	2	2	0.000292	0.000024	0.000316	40
7	3	1	0.000303	0.000157	0.00046	40
7	4	0	0.001353	0.001093	0.002446	40

8	1	4	0.027213	0.000001	0.027214	92
8	2	3	0.006201	0.000002	0.006203	92
8	3	2	0.002536	0.000008	0.002544	92
8	4	1	0.003945	0.000067	0.004012	92
8	5	0	0.033687	0.000471	0.034158	92
9	1	5	0.536247	0.000002	0.536249	352
9	2	4	0.212134	0.000051	0.212185	352
9	3	3	0.072894	0.000438	0.073332	352
9	4	2	0.034015	0.004014	0.038029	352
9	5	1	0.076316	0.036051	0.112367	352
10	1	6	10.812877	0.000001	10.812878	724
10	2	5	4.473587	0.000003	4.47359	724
10	3	4	1.954345	0.000019	1.954364	724
10	4	3	0.768767	0.000175	0.768942	724
11	1	7	364.637563	0.000001	364.637564	2680
11	2	6	155.441078	0.000093	155.441171	2680
11	3	5	87.838195	0.001001	87.839196	2680
11	4	4	34.876625	0.010997	34.887622	2680
12	4	5	1266.992875	0.000431	1266.993306	14200

Figure 3: Second Optimization- Sum on CPU

N	numBX	numGen	numBY	GPU Time	CPU Time	Total Time	Solutions
4	1	0	1	0.000045	0.000001	0.000046	2
5	1	1	1	0.000057	0.000001	0.000058	10
5	1	1	2	0.000052	0.000001	0.000053	10
5	1	1	3	0.000054	0.000001	0.000055	10
5	1	1	4	0.000051	0.000001	0.000052	10
5	1	1	5	0.000050	0.000001	0.000051	10
5	2	0	1	0.000048	0.000002	0.000050	10
6	1	2	1	0.000180	0.000001	0.000181	4
6	1	2	2	0.000117	0.000001	0.000118	4
6	1	2	3	0.000096	0.000001	0.000097	4
6	1	2	4	0.000086	0.000001	0.000087	4
6	1	2	5	0.000077	0.000001	0.000078	4
6	1	2	6	0.000075	0.000001	0.000076	4
6	2	1	1	0.000066	0.000001	0.000067	4
6	2	1	2	0.000063	0.000001	0.000064	4
6	2	1	3	0.000078	0.000002	0.000080	4
6	3	0	1	0.000104	0.000003	0.000107	4
7	1	3	1	0.001898	0.000001	0.001899	40
7	1	3	2	0.001038	0.000001	0.001039	40
7	1	3	3	0.000771	0.000001	0.000772	40

7	1	3	4	0.000618	0.000001	0.000619	40
7	1	3	5	0.000579	0.000002	0.000581	40
7	1	3	6	0.000578	0.000002	0.000580	40
7	1	3	7	0.000590	0.000002	0.000592	40
7	2	2	1	0.000271	0.000003	0.000274	40
7	2	2	2	0.000297	0.000003	0.000300	40
7	3	1	1	0.000287	0.000008	0.000295	40
7	3	1	2	0.000444	0.000013	0.000457	40
7	4	0	1	0.001418	0.000041	0.001459	40
8	1	4	1	0.025267	0.000001	0.025268	92
8	1	4	2	0.013838	0.000001	0.013839	92
8	1	4	4	0.009458	0.000000	0.009458	92
8	1	4	6	0.007703	0.000001	0.007704	92
8	1	4	100	0.007689	0.000012	0.007701	92
8	2	3	1	0.005653	0.000001	0.005654	92
8	2	3	2	0.004858	0.000002	0.004860	92
8	2	3	4	0.004532	0.000004	0.004536	92
8	2	3	20	0.004587	0.000019	0.004606	92
8	2	3	200	0.011102	0.000170	0.011272	92
8	3	2	1	0.002324	0.000007	0.002331	92
8	3	2	2	0.002639	0.000015	0.002654	92
8	4	1	1	0.004074	0.000060	0.004134	92
8	4	1	2	0.007534	0.000110	0.007644	92
8	5	0	1	0.034607	0.000486	0.035093	92
9	1	5	1	0.506303	0.000000	0.506303	352
9	1	5	2	0.330546	0.000004	0.330550	352
9	1	5	10	0.314289	0.000002	0.314291	352
9	1	5	1000	0.317009	0.000180	0.317189	352
9	2	4	1	0.198608	0.000004	0.198612	352
9	2	4	2	0.192335	0.000004	0.192339	352
9	2	4	4	0.175264	0.000009	0.175273	352
9	3	3	1	0.067708	0.000017	0.067725	352
9	3	3	2	0.068587	0.000031	0.068618	352
9	4	2	1	0.032422	0.000124	0.032546	352
9	4	2	2	0.039410	0.000282	0.039692	352
9	5	1	1	0.082850	0.001133	0.083983	352
9	5	1	2	0.160625	0.002225	0.162850	352
10	1	6	1	10.268869	0.000000	10.268869	724
10	1	6	2	8.876146	0.000001	8.876147	724
10	1	6	100	7.458255	0.000024	7.458279	724
10	1	6	1000	7.445729	0.000167	7.445896	724
10	1	6	5000	7.478029	0.000823	7.478852	724
10	2	5	1	4.229290	0.000002	4.229292	724

10	2	5	2	3.978458	0.000003	3.978461	724
10	2	5	100	3.269084	0.000166	3.269250	724
10	3	4	1	1.837538	0.000018	1.837556	724
10	3	4	2	1.839613	0.000036	1.839649	724
10	4	3	1	0.723600	0.000166	0.723766	724
10	4	3	2	0.739102	0.000331	0.739433	724
11	1	7	1	346.812844	0.000001	346.812845	2680
11	1	7	5000	247.414000	0.001337	247.415337	2680
11	2	6	10000	247.574000	0.002701	247.576701	2680
11	2	6	1	147.404750	0.000005	147.404755	2680
11	2	6	1000	134.181125	0.002822	134.183947	2680
11	2	6	5000	134.983422	0.014293	134.997715	2680
11	3	5	1	83.203781	0.000034	83.203815	2680
11	3	5	100	83.578570	0.003000	83.581570	2680
11	4	4	1	32.946035	0.000357	32.946392	2680
11	4	4	10	33.443043	0.003289	33.446332	2680
12	4	5	1	1209.907625	0.000412	1209.908037	14200
12	4	5	1000	1261.401875	0.410114	1261.811989	14200

Figure 4: Second Optimization- Sum on GPU

N	numBX	numGen	numBY	GPU Time	Solutions
4	1	0	1	0.000047	2
5	1	1	1	0.000057	10
5	1	1	2	0.000060	10
5	1	1	3	0.000067	10
5	1	1	4	0.000075	10
5	1	1	5	0.000081	10
5	2	0	1	0.000080	10
6	1	2	1	0.000181	4
6	1	2	2	0.000116	4
6	1	2	3	0.000097	4
6	1	2	4	0.000087	4
6	1	2	5	0.000089	4
6	1	2	6	0.000099	4
6	2	1	1	0.000097	4
6	2	1	2	0.000152	4
6	2	1	3	0.000219	4
6	3	0	1	0.000397	4
7	1	3	1	0.001899	40
7	1	3	2	0.001038	40
7	1	3	3	0.000772	40
7	1	3	4	0.000618	40
7	1	3	5	0.000577	40

7	1	3	6	0.000581	40
7	1	3	7	0.000590	40
7	2	2	1	0.000905	40
7	2	2	2	0.001676	40
7	3	1	1	0.000905	40
7	3	1	2	0.001676	40
7	4	0	1	0.005736	40
8	1	4	1	0.025268	92
8	1	4	2	0.014803	92
8	1	4	4	0.009458	92
8	1	4	6	0.007709	92
8	1	4	100	0.008485	92
8	2	3	1	0.005655	92
8	2	3	2	0.004839	92
8	2	3	4	0.004412	92
8	2	3	20	0.006907	92
8	2	3	200	0.033998	92
8	3	2	1	0.003036	92
8	3	2	2	0.004374	92
8	4	1	1	0.011449	92
8	4	1	2	0.022286	92
8	5	0	1	0.093512	92
9	1	5	1	0.506266	352
9	1	5	2	0.330562	352
9	1	5	10	0.318296	352
9	1	5	1000	0.338037	352
9	2	4	1	0.209251	352
9	2	4	2	0.187737	352
9	2	4	4	0.179539	352
9	3	3	1	0.082478	352
9	3	3	2	0.086391	352
9	4	2	1	0.049798	352
9	4	2	2	0.071729	352
9	5	1	1	0.217038	352
9	5	1	2	0.427807	352
10	1	6	1	10.268467	724
10	1	6	2	8.876130	724
10	1	6	100	7.461655	724
10	1	6	1000	7.443640	724
10	1	6	5000	7.582830	724
10	2	5	1	4.517959	724
10	2	5	2	4.163708	724
10	2	5	100	3.669357	724

10	3	4	1	2.066147	724
10	3	4	2	2.071890	724
10	4	3	1	0.866400	724
10	4	3	2	0.904880	724
11	1	7	1	346.428875	2680
11	1	7	5000	247.484703	2680
11	2	6	10000	247.857484	2680
11	2	6	1	147.295000	2680
11	2	6	1000	134.509078	2680
11	2	6	5000	136.907203	2680
11	3	5	1	82.908219	2680
11	3	5	100	83.629305	2680
11	4	4	1	32.536514	2680
11	4	4	10	33.363012	2680

Figure 5: Consider all combinations of placement of queen in first column

N	numBX	numGen	numBY	GPU Time	CPU Time	Total Time	Solutions
4	1	0	1	0.000042	0.000000	0.000042	2
5	1	1	1	0.000060	0.000001	0.000061	10
6	1	2	1	0.000215	0.000001	0.000216	4
7	1	3	1	0.002353	0.000001	0.002354	40
8	1	4	1	0.030703	0.000001	0.030704	92
9	1	5	1	0.741323	0.000002	0.741325	352
10	1	6	1	16.268259	0.000002	16.268261	724
10	4	3	1	1.678810	0.000363	1.679173	724