CUDA Picture Cross Solver

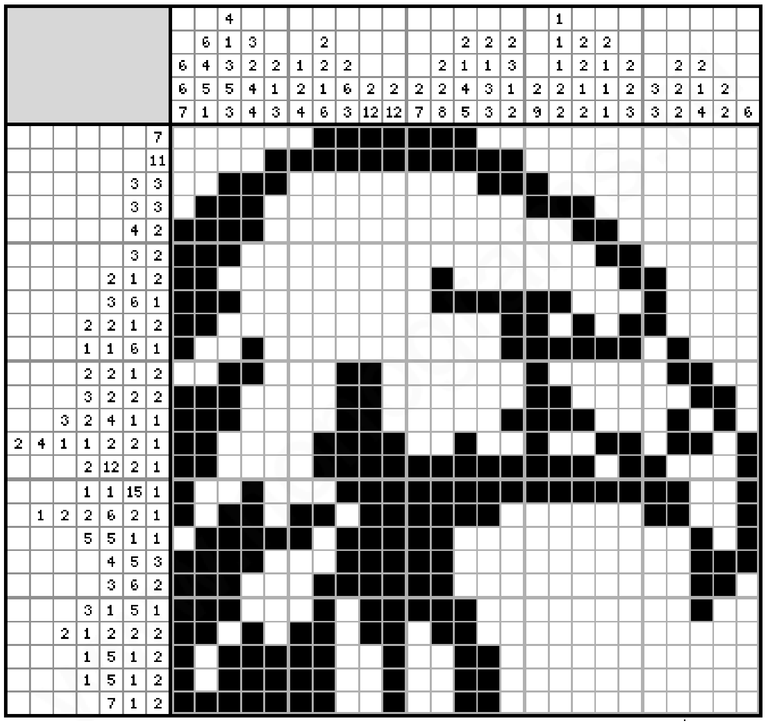
Goran Goran (ggoran) and Yunchieh Janabelle Wu (yunchiew)

Summary

Picture Cross is a nonogram solver that solves grid puzzles with hints on each column and row. Each puzzle only has one solution and the solver is required to look ahead with a depth first search (DFS) algorithm. Through our experiments, we found that our parallel implementation running on RTX 2080 B GPUs is better in performance than the sequential implementation. Our parallelized implementation splits up the calculation of the permutations into different GPU cores. With all the permutations calculated, we split up the DFS algorithm into different GPU cores to run in parallel. More about the algorithm in the later sections of this report.

Background

Picture Cross is a grid and logic game that shades in cells depending on the hints given at each row and column. This game is essentially a nonogram puzzle. The objective of the game is to determine which cells are shaded in based on these hints. These hints are a sequence of numbers, and each of these numbers represent the number of contiguous regions of shaded cells in that row or column. The spacing between each of the contiguous blocks can be 1 or more unshaded cells. Below is an illustration of one of the puzzles and solutions of the Picture Cross game.



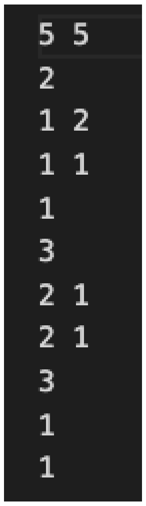
Data Structures

For our Picture Cross solver implementation, we made use of several key data structures. Our main data structure, pic_cross_t, consisted of our hints, the puzzle, and the two dimensions of our puzzle. Our hint member contained information about the blocksizes for each row and column, and was read from our input file. The hints member in our sequential implementation took the form of a 2d vector of int vectors, this was because we needed an unbounded dynamic array to store our hints since the size was variable. This was slightly harder to implement in CUDA due to its lack of the std library. We ended up making use of the Thrust CUDA C++ template library. Thrust is a C++ template library for CUDA based on the Standard Template Library (STL). Thrust allowed us to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C. So our hint member now took the form of 2D thrust device vector of device vector ints. The rest of the members of our pic_cross_t struct were much simpler in comparison, the puzzle member simply took the form of a 2D array of ints, and the dimensions of the puzzle were stored as ints. The puzzle member allows us to store all of the information we need to draw and describe our Picture Cross solution, and partial solutions.

Input and Outputs

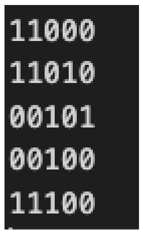
The inputs into our algorithm consist of the dimensions of our unfilled puzzle (number of rows/cols), along with a list of hints which consist of several distinct block sizes for the picture cross puzzle. Our output is a filled row by col size board where the filled in blocks are represented as 1's and the unfilled blocks are represented as 0's.

Example Input (5x5):



```
5 5
2
1 2
1 1
1
3
2 1
2 1
3
1
1
```

Example Output:



```
11000
11010
00101
00100
11100
```

Computation Expensive

The very nature of picture cross puzzles, makes them all not extremely computationally expensive. Since picture cross puzzles are meant to be human solvable and only have one correct answer, it's very likely that the portions we parallelize will not see very large benefits. The portions of our picture cross algorithm which seem like they may benefit the most from parallelization are our calculations of the row permutations and the depth first search. The row permutation parallelization is fairly straightforward as there are no shared memory values, and now way for calculations between rows to affect each other. The depth first search is computational expensive, but is slightly more difficult to parallelize due to its recursive nature relying on values from previous runs. As the size of the board grows larger

Workload

The workload of our solver consists of the previously mentioned row permutation calculations, along with the DFS search algorithm to construct our final puzzle. How this works is that the hints are passed through the calculations of the row permutations to figure all possible row solutions. These row permutations are stored in a 2D array. We then take all these possible permutations and perform a DFS algorithm on it. Within the DFS (depth first search) we need to check that our current solution we have picked is a valid solution. We check the validation of the columns by keeping track of two 2D arrays. We have a 2D array called colVal. At every cell in the array, the value represents the current position within the current block. Block is the contiguous region of cells in each row and column. On the other hand, we have another 2D array called colIx that represents the current block index. If we are on the second block of that column, then the value at that cell will be 1. These arrays are used to check if the puzzle still satisfies all the row and column hints that were defined and populated in the pic_cross_t data structure. These are the usage of our data structures in the sequential program. The same operations on these sequential data structures are also performed on the CUDA data structures, 2D thrust device vectors of ints.

There is potential that our algorithm can benefit from SIMD execution, as that will allow us to check all of our row permutations against our marks and val array to see which of them are valid consecutive row placements. Since the data is stored as bitmasks and there is not much need for communication, this execution lends itself very well to a CUDA implementation.

Approach

During our brainstorming sessions for our approach we thought about the multiple different technologies that we used/learned about in class, and we finally settled on using GPUs for our parallelization. We started with a simple C++ sequential Picture Cross implementation and slowly converted it into CUDA format for our parallelization scheme. We targeted the GHC machines containing NVIDIA GeForce RTX 2080 B GPUs, but were slightly limited by the fact that each cluster machine only contained a single GPU.

Parallelism in Row Permutations

The most obvious place to start in parallelizing our picture cross solver, is in calculating our row permutations. At the beginning of our solver algorithm, we precalculate all of the possible puzzle rows without any regard for actual correctness. These row permutations are only based off of the current row's hints, and have no effect on the values of any other row permutation or any other shared memory variable. This means that no synchronization was needed between threads after calculating these row permutations, and we should ideally see high speedups compared to a single threaded cpu sequential implementation. Since picture cross puzzles are made to be human solvable, the number of possible permutations per row is relatively small, meaning we will not run into issues of lack of memory space.

Parallelism in DFS

Another focus for our parallelization was in our DFS (depth first search) algorithm. How this algorithm worked was it searched through a tree of our row permutations, and took a guess at every node and continued recursively until it created a not valid puzzle. Our approach was to spawn new threads at every level for every possible row permutation. These threads would then continue diving lower and lower into our tree until it has either come up with a valid puzzle or failed. We eventually ran into issues where we were consuming too much memory with so many threads running. In order to prevent too many threads from being spawned and us running into memory issues, the number of active threads was capped. In the case that the thread fails, it will simply return, and allow for more threads to be spawned at other levels. If a thread succeeds, a shared variable between all threads that is being constantly read will be updated and all other threads will be told to return without writing their puzzle. The valid thread will then write its result into shared memory before returning. This worked relatively well for smaller board sizes, but as we approached larger boards we began to run into issues of running out of shared block memory space.

Parallelism DFS without saving boards

After realizing that our parallelism scheme for DFS was not viable for larger boards we attempted to rework how we are using threads to parallelize our validation scheme. We noticed that among our larger boards, the most noteworthy and commonly appearing block size was 1, and by a vast margin. So we decided to pivot our approach and use parallelism to narrow down the possible valid puzzle combinations in our working tree. Now instead of having threads save their entire working puzzle to their shared block memory, we will have that only consider a sequence of two rows at a time. Each thread will consider the current row permutation along with a permutation of the row before it, and check if these two rows are valid consecutive rows.

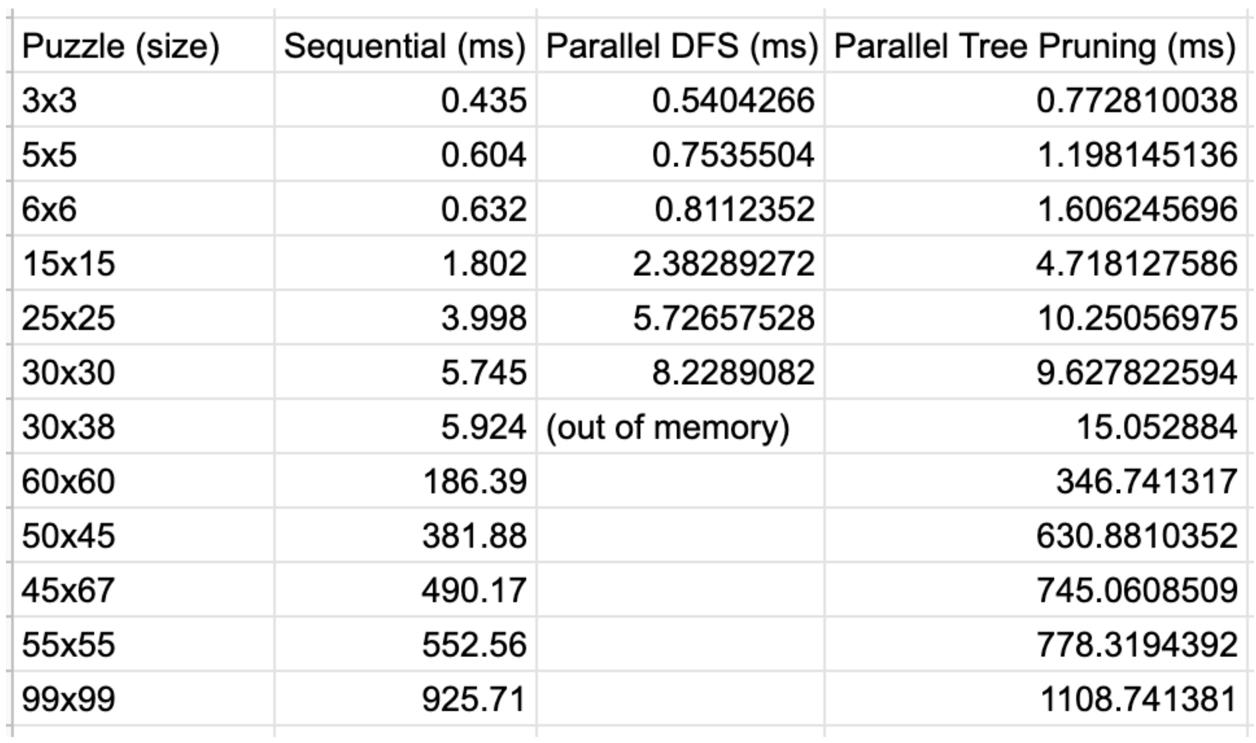
A global array is then updated at the index corresponding to that particular row, that contains an array of the length of all current permutations of the next row. Since no thread is ever updating an index that does not belong to it, there is no need for synchronization in our implementation. We then will run our sequential implementation of DFS on the hopefully much smaller tree of possible puzzle possibilities and output the result.

Results

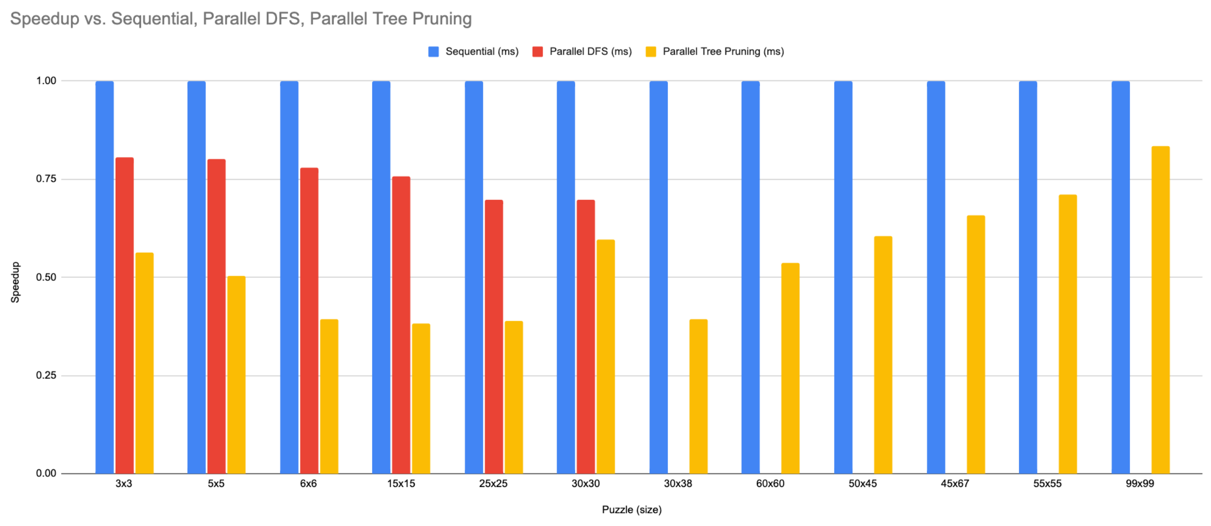
Experimental Setup

Our experiment is running based on the time it takes to run through both preprocessing and the main chunk of the DFS calculations. The unit of the time is measured in milliseconds. Sequential is the regular C++ implementation with no extra parallelism included. For the parallel DFS, we are parallelizing our DFS algorithm. Lastly the difference between the 2 parallel implementations is the pruning that happens in the third column to avoid reaching maximum memory on these GPUs. Each of the puzzles vary in size and difficulty. Most of the time the complexity of a puzzle increases with the size of it. In our graph below, we will be showing the speeedup using the sequential implementation as the relative/baseline performance.

Data



Puzzle (size)	Sequential (ms)	Parallel DFS (ms)	Parallel Tree Pruning (ms)
3x3	0.435	0.5404266	0.772810038
5x5	0.604	0.7535504	1.198145136
6x6	0.632	0.8112352	1.606245696
15x15	1.802	2.38289272	4.718127586
25x25	3.998	5.72657528	10.25056975
30x30	5.745	8.2289082	9.627822594
30x38	5.924	(out of memory)	15.052884
60x60	186.39		346.741317
50x45	381.88		630.8810352
45x67	490.17		745.0608509
55x55	552.56		778.3194392
99x99	925.71		1108.741381



Analysis

In our analysis we compared our three different implementations, our sequential algorithm, our parallel DFS algorithm, and our parallel tree pruning algorithm. Our sequential algorithm (blue) works much the same as described in the background, with row permutation calculations followed by a DFS algorithm. Our parallel DFS algorithm (red) works the same as described in the approach, and you can see from the data that it fails on puzzles larger than 30x30. Our last algorithm, the tree pruning (yellow), works the same as described in the approach section.

We were unable to obtain any sort of measurable speedup in either of our parallel implementations, for two main reasons. One the lack of computation workload that can be parallelized resulting in high overhead, and just sheer memory limitations. Since picture cross puzzles are made to be human solvable, and have only one real solution, its very nature makes it not very compute intensive. Although we would spawn threads to traverse every possible combination of our row permutations, this turned out to be not that useful since many of them would simply immediately terminate. The overhead from creating these short lived threads resulting in a much higher overhead, at least for the parallel DFS implementation. On the other hand, in our parallel tree pruning algorithm, each thread seemed to be able to achieve useful work as each computation resulted in our set of possible board solutions constantly shrinking. We can see from our graphs that although the overhead still seems to cause it to have no actual speedup compared to the sequential implementation, that as the board sizes grow and the number of blocks sizes of one increase, speedup increases as well. We can theorize that for larger boards, above 99x99 we may be able to obtain actual speedups. Sadly, due to lack of availability of these larger puzzles, we were not able to put this theory into practice.

Another limitation in our abilities to see any actual speedup resulted from memory limitations. The parallel DFS algorithm simply failed on larger boards since there was not enough space in the shared block memory to store all of the puzzles that its threads were constantly computing, something we had not considered when we first started working on this approach. It is possible that on larger GPUs with more shared space we could potentially see actual speedups in our parallel DFS algorithm, as overhead becomes increasingly smaller and smaller. Another memory related issue we ran into was memory latency. Although we tried to limit the transfer of memory as much as we possibly could, it was still often necessary to transfer memory between global data variables and shared memory. These data transfers were also at variable times for small amounts of data, if we were able to lump these memory transfers into a larger lump we would be better able to take advantage of the GPU to CPU PCIe lane buses. After looking at the Nvidia Visual Profiler, for many of our puzzles we were only about to see ≈12% transfer efficiency.

In hindsight after running into a lot of memory issues, and the surprisingly high overhead of our CUDA threads, it may have made more sense to use CPUs, at least for the larger puzzles. We might have been able to chop up the puzzle into larger distinct and done more work per core, and limited the communication between the CPU's as much as possible. That way we may have been able to see an increase in speedup.

References

We first found the Wikipedia page (https://en.wikipedia.org/wiki/Nonogram) about nonograms to give us a good idea and overview of what it is and how to solve it. We used a few resources to help us with the approach of our sequential solver. There are a few resources that use Python (https://github.com/mprat/nonogram-solver) and JSON (https://github.com/ThomasR/nonogram-solver) to solve the picture cross puzzles.

Distribution of Work

Work distribution was 50%/50% by both partners, all work was done together throughout each portion.