A CUDA Accelerated Parallel Nonogram Solver

Benjamin Huang (zemingbh) Eric Sun (ehsun)

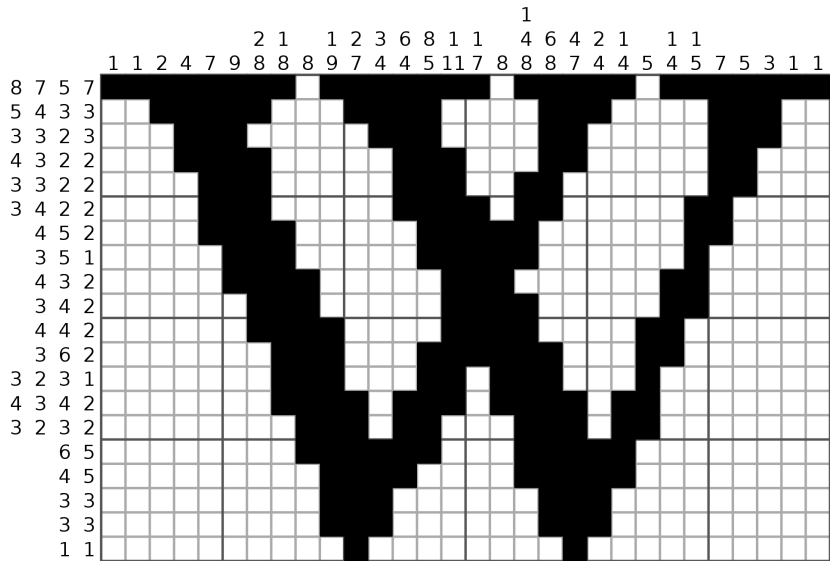
December 2018

Abstract

We implemented a GPU-accelerated parallel nonogram solver that can find solutions for puzzles that require lookahead solving. We found that a sequential implementation is faster solving puzzles up to and including the largest size that humans typically solve (99x99). While it's possible that a parallel implementation would have the advantage for larger puzzles, it's possible that such puzzles would exceed the resources available on most GPUs.

1 Background

Nonograms are logic puzzles built around a rectangular grid divided into cells. The solver's objective is to determine, for each cell, whether the cell is shaded or not. The constraints on the shading of cells are given as number sequences on each row and column, indicating to the solver the number of contiguous regions of shaded cells in that line (row or column) and the number of shaded cells in each contiguous region. An example is shown below:



1.1 Data structures

The state of the cells of a nonogram can be represented by a matrix or board whose elements can take one of three different values: white (unfilled), black (filled), or unknown. For each row and column there is an associated set of constraints, which can be represented as a list of numbers.

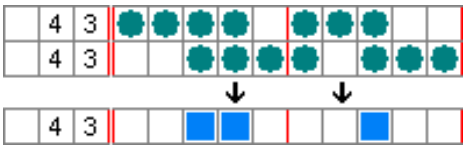
Note that while the goal is a board with no unknowns that satisfies the associated constraints, it is possible that a particular board will contradict its constraints or be in such a state that a contradiction is implied in the future. This will be explained in more detail below.

1.2 Operations

Solving nonograms is an iterative process. Any given cell being determined means that other cells in its row and column can also possibly be determined with the new information. There are two basic kinds of operations involved in solving nonograms: simple solving, and lookahead solving.

1.2.1 Simple solving

One key operation involved in solving nonograms is advancing the state of any particular row or column, which will just be referred to as a line. Given a line and its list of constraints, a simple solve operation will (potentially) be able to change at least one unknown cell to either filled or unfilled. There are a variety of algorithms that can accomplish this, the simplest of which is illustrated below:



In this example, consider a single line with runs of length four and three. By examining the both the leftmost and rightmost arrangements of the runs, it is apparent that in all cases the cells marked in blue will be filled. In this case, it is not also guaranteed that the cells in white are unfilled; those cells will still be unknown. However, if the first column of the puzzle is solved and it is determined that the first cell of the row is filled, then there is enough information to completely solve the row.

Note that many nonograms can be solved by iterated simple solving. By repeatedly alternating between applying simple solvers to the rows, and to the columns, more and more cells will be determined until eventually the entire puzzle is solved.

This iterated simple solving method can be thought of as a single step in which a board is reduced to its most solved form (fewest unknown cells).

1.2.2 Lookahead solving

For some puzzles, applying simple solving alone is not enough to find a solution. It is possible to get to a state where no additional cells can be determined from the known information. In this case, it is necessary to make a guess about an unknown cell. The process is relatively simple: a guess is made, and then the board is solved with the guess assumed to be true. There are then three possibilities:

1. A valid solution is found, in which case the guess was correct.
2. A contradiction is found, meaning the guess was incorrect.
3. The board is again unable to advance, meaning another guess must be made.

It is clear that lookahead solving is a traditional backtracking algorithm. As with any backtracking problem, the cell and value of the guess is irrelevant to eventually finding a solution—eventually a solution will be found. However, to maximize performance various heuristics can be used to make a guess that is either likely to be correct, or likely to arrive at a contraction quickly, or both.

Note that lookahead solving does affect the correctness of a board while simple solving does not. We consider a board to be correct if there exists some way to fill in all of its unknown cells such that the constraints are satisfied. It is possible for a board to be incorrect and unsolved: this is a board that is eventually guaranteed to have a contradiction after it is solved more. However a board can only become incorrect through guessing. Solving a puzzle starts with the board full of unknowns which is trivially correct.

1.3 Inputs and outputs

The input to the algorithm is the set of constraints. The board starts in an entirely unknown state.

The output of the algorithm is a board in which every cell is either filled or unfilled, that also satisfies all of the constraints given in the input

1.4 Computational expense

Typically nonograms are meant to be solved by humans. This means that they tend to have relatively low problem sizes. There are two main parameters involved in determining the total size of the problem: number of constraints per line, and size in rows/columns.

Typically puzzles will have on the order of 10 or fewer constraints per line. Because many nonograms form basic images, lines with fewer constraints are more frequent. Puzzles will also be relatively small in grid size. The largest ones are on the order of 100x100, meaning there are only 200 total lines.

This tendency to have small puzzles means that any results need to be contextualized. When solving a puzzle meant for humans it's likely that the overhead of any parallelism will dominate.

1.4.1 Simple solving

Simple solving techniques are relatively expensive individually. They involve iteratively working through the possibilities for a given line until unknown cells can be filled in. The amount of work involved in this process increases as the number of runs in a given line increases, or as the length of the line increases.

In total the simple solving techniques are also expensive because they must be repeatedly applied to every row of the board, as well as every column. As the board grows the expense of running all the solvers increases.

1.4.2 Lookahead solving

The process of lookahead solving outside of simple solving is not computationally expensive. If heuristics are used they can be more expensive depending on which algorithm is chosen.

1.5 Workload

The workload for solving a puzzle consists of iterated simple solving and lookahead solving. Within simple solving there are dependencies between the row steps and column steps.

1.5.1 Parallelism

Simple solving techniques are applied to a single line (row or column) at a time. They read only the constraints for that line, as well as the contents of the cells in the line, and they write only to cells in the line. Thus simple solving on either every row or every cell can be done in parallel both computationally, and data-wise.

Because the results from solving the rows are necessary for solving the columns and vice versa, solving rows and columns in parallel is possible but could potentially waste compute resources.

1.5.2 Locality

There is spatial locality as the simple solver will need to iterate over the lines. When the lines are rows this allows us to exploit cache locality, and storing the board in a column major format can allow us to potentially use the same locality for the columns.

1.5.3 SIMD

Depending on how simple solvers are implemented it is possible that SIMD execution could speed up the simple solvers. In particular, a solution that is largely branchless would be able to take advantage of SIMD. This technique would also likely lend itself well to GPU/CUDA execution because of both the SIMD possibilities and the fact that each computation requires very little data, reducing communication overhead.

2 Approach

Two different solvers were written: a sequential solver and a parallel solver. The sequential solver used many of the same techniques as the parallel solver, but ran on a single CPU core. It is not described in detail.

The parallel solver was broken down into the components described in the background section. The goal was to pick a set of technologies that fit the problem and the proposed solution best, and then to tune the specific algorithms and data structures.

2.1 Technologies

A number of technologies were considered, however eventually GPUs/CUDA were selected as the platform for the solver. There were a few primary reasons for this decision.

One was that the computations involved in solving nonograms are not memory-intensive. As mentioned above, for any given board the constraints need to be stored, as well as the actual state of the board. For a board with a reasonable number of constraints per line (on the order of 10) that has N rows and M columns, the amount of memory required is O(NM) with a relatively small constant that are implementation dependent. In particular, this means that shared block memory is large enough to hold the state of a board.

In addition to having a memory model that maps well to solving nonograms, CUDA's computation model is also a good fit. A large number of threads (1024) can fit into a thread block, meaning each individual line solver can run in parallel even for larger puzzles, and they can all share the same block memory.

Worth noting is that as puzzles get larger they outgrow shared block memory into global memory, which incurs a performance penalty. At a certain size the number of threads will exceed the maximum for a block as well.

There were also some downsides to using CUDA that were discussed in the decision process. One is that its SIMD capabilities went unused, because the simple solvers running in parallel diverge too much. Another is that it would be difficult to test very large problem sizes because the Gates machines only have a single GPU each.

Another option that was considered was the Xeon Phi machines, because they would better handle divergent execution, however it was ultimately decided that having the higher number of CUDA threads would be more beneficial.

2.2 Problem mapping

The problem maps onto a GPU roughly as described in the above section. There are two regions of shared block memory that hold the entire board, one in row-major format and one in column-major format. Additionally, there is a region of shared block memory for the individual line solvers to hold their data. When the simple solvers run they are all mapped along one dimension of the same block.

In addition, there are parts of the solving algorithm, particularly the lookahead solving, as well as memory operations that need to only run once. These are run from the first thread (chosen arbitrarily).

A goal when the solution was developed was that as much of the solver run on the GPU as possible. A significant portion of overhead from using CUDA comes from copying memory back and forth between the host and the device. Rather than do that, or create a number of additional kernels to perform specific operations on the data, the majority of the algorithm runs as a single kernel.

No big changes were necessarily needed to enable better mapping to the parallel resources. The most parallelizable element of the system, the simple solvers, ran independent of each other in the sequential implementation already.

2.3 Optimization iterations

There were a few important optimizations that were arrived at by iterating. They included reducing dependencies between the row solvers and the column solvers, adding heuristics, and implementing lookahead solving.

2.3.1 Solver dependencies

One optimization that was found was running both the row and column solvers at the same time. Because the line solvers were written to be independent, they were eventually modified so that they only time they wrote data to the board in shared block memory was when they had a new correct value to write.

The fact that every value being written was guaranteed to be correct also means that there is no synchronization necessary if/when multiple solvers write to the same cell. They will either write the same value, in which case the value will be written one way or another, or they'll write different values. If they write different values then the board must have been incorrect, in which case there will be a contradiction no matter which value ended up actually getting written.

Another related optimization that was mentioned above was storing two copies of the board data, one in row-major and one in col-major format. It was inspired by the assignments during the semester where such a strategy was a way to improve locality and therefore cache-friendliness of the data.

In this case, it works particularly well because each line solver reads and writes only its particular line data. Additionally, writes are much less frequent because they only occur when correct values have been found. Compounding that is the fact there is often dependence between row solving and column solving that does not prevent them from running at the same time, but that limits performance improvement.

Consider the case where no new cells can be solved by the row solvers, but some can be by the column solvers, and both run at the same time. Then only the column solvers will update the board. On the next iteration, only the row solvers (if any) will update the board, and so on. This dependence means that

while it's possible for the solvers to all run in parallel, the actual benefits are relatively minimal.

2.3.2 Heuristics

To assist lookahead solving, a heuristic was used to determine which cell to make a guess in instead of guessing blindly. The heuristic works as follows, for each cell X:

1. Check if X is between an white cell and an unknown cell, either in its row or its column.
2. If it is, determine the number of cells in X_{row} and X_{col} directly affected if it is filled black. This is also the minimum length constraint that can still be matched by X. The number of cells is X_{score} (X_{score} for the cells that do not satisfy (1) is -1).
3. Determine the maximum score across the entire board. This cell is selected to guess.

The heuristic works based on the idea that X is in a corner and the adjacent solved sides extend fairly far down the line. When X is filled, it causes a block of adjacent cells to also be filled both in the row and the column. The adjacent cells are likely to be edge cells, which will cause more blocks to fill, but this time only in either the row or the column. This quickly causes contradictions in the unsolved lines adjacent to X. We want to cause as much havoc as possible in the board, so we pick X such that it affects the most cells when filled in, hoping to quickly arrive at a contradiction and confirm X as unfilled.

The nature of this heuristic is also that it is extremely parallelizable. Determining the score for each cell is done independently of all the other cells, and is spatially local to the cell itself. The line solvers keep a record of the possible run positions, so the cells also only access their respective row and column solvers to determine their score. Finally, a reduction is performed on the entire board.

2.3.3 Lookahead solving

When both guesses of lookahead solving reach an unknown state (no contradiction) it is not always necessary to completely discard the results. By comparing the two guesses' results, the cells which are common to both guesses are certain and can be updated on the master board.

In cases when there are no common cells and no progress can be made on either guess (black or white), then there are two options: either make a second guess from the states created by the first guess, or cancel the first guess and make a different guess. Because making further guesses (going depth-first) requires a lot of memory to track state and we are limited on GPU shared memory, we opt to use breath-first instead. First, this allows the heuristic previously computed to be reused after discounting the cell used in the first guess.

Also, when solving with a guessed cell (a hypothetical state), a single thread block is used. The hypothetical board is kept only in the block's shared memory, and eventually, only modified cells are eventually written back to the master board in global memory if necessary. In case of a contradiction, the shared memory can simply be discarded. This helps to reduce data transfer to and from the GPU.

3 Results

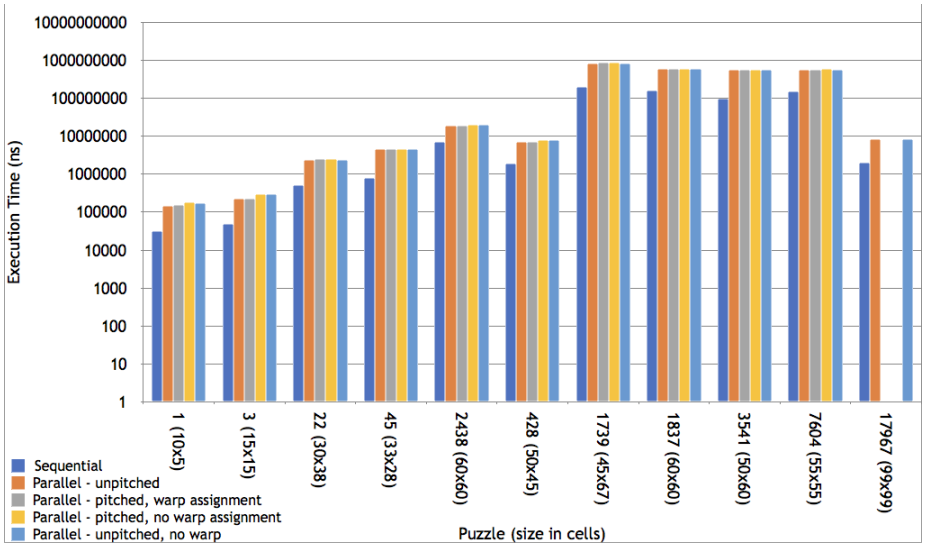
Ultimately a speedup was not realized on any of the puzzles the solvers were benchmarked with. There are a number of factors that could have contributed to this result; they are outlined below. While in theory a parallel implementation could outperform a sequential one, in practice it is likely that such a result could be achieved using the same technologies that this solver was created with.

3.1 Setup

Time in nanoseconds was measured for execution of both the parallel and sequential implementations. For the parallel implementation, time to transfer data to device memory was not included in the measreument. A number of puzzles were tested. They varied in size as well as complexity, or more measurably, how many guesses were required to solve the puzzle.

3.2 Data

			Parallel			
Puzzle	Guessing	Sequential	Unpitched	Pitched, warp assignment	Pitched, no warp assignment	Unpitched, no warp
1 (10x5)	No	32495	147133	158400	178313	177412
3 (15x15)	No	48803	226853	233402	305115	301354
22 (30x38)	Once	520410	2444099	2463431	2485525	2457599
45 (33x28)	Twice	816032	4687777	4733862	4731041	4690133
2438 (60x60)	No	7311862	19107473	19135279	20428909	20396841
428 (50x45)	Once	1960037	7218986	7266577	7830230	7780099
1739 (45x67)	Many	206231844	832682212	886362668	897340535	857788874
1837 (60x60)	Many	158422450	589038618	593398377	591757435	589738732
3541 (50x60)	Many	99856107	570744325	574033813	573329374	568743937
7604 (55x55)	Many	151717386	569611214	574630976	588833255	583794084
17967 (99x99)	No	1994000	8343894			8456278



3.3 Analysis

3.3.1 Memory

One of the main limiting factors was memory latency. While the transfer of data was minimized as much as possible during the solving process (transfer before and after solving was excluded from the timings), it was still necessary to transfer data between device global memory and shared memory. For the simple boards that could be solved by pure line solving, this transfer only happened once, however, the solving was also quickly finished, so the overhead of data transfer was significant.

On larger boards, every time a guess was made, new kernels were launched, with the need to copy to shared memory again (and the need to copy back depended on whether a contradiction was reached). One of the goals for a lookahead solver is to find contradictions quickly. However kernels that found contradictions, as well as kernels with bad guesses that made no progress, were relatively short-lived. This meant that the data transfer for that kernel was essentially wasted.

One of the attempted optimizations was to set the board pitch to a power of two to reduce bank conflicts. However, instead of reducing time taken, this slightly increased time taken. The only likely reason for this is that the additional memory allocation required caused a slowdown. This is evidenced by the much large increase in computation time for #1739 when going from unpitched to pitched, due to the column width being 67 (just above 64) which would result in a pitch of 128 for the column-major array. Compare this to #1837, #3541 and #7604 which are all just under 64 in both dimensions, and the difference in their computation time between pitched and unpitched is significantly smaller. Using NVVP (the NVIDIA Visual Profiler) also suggested that cudaMalloc was

eating up a large amount of time.

The amount of data being transferred was very small compared to typical transfers for a GPU. NVVP reported only a 17% memory transfer efficiency on #1739. This is hard to address since nonograms are small. Data transfer was already limited as much as possible even between shared and global device memory and global device and host memory, but often it was still necessary to set flags to tell the CPU about the results of the kernel to make decisions about what to do next. These small (but necessary) data transfers would have incurred a large overhead.

Finally, though shared block memory was large enough (48KB) for a single board, it was not large enough for more than two boards at once. This meant that for lookahead solving, it was not possible to make more than two guesses (both on the same cell) and run in parallel without having to use global memory. Using global memory was not desirable since a copy of the board would still have to be made, which would mean allocation, data transfer and eventually freeing the memory, incurring more overhead.

3.3.2 Divergence

As hypothesized, another major problem was divergent execution. In order to solve efficiently, it was necessary to condition on the state of each line at many points, and every line was in a different state. Some lines would be solved and some would not, and some lines would have many constraints and some would have only a few. NVVP shows a warp execution efficiency average of 25% on #1739.

One of the optimizations in this area was to split the row and columns solvers into different warps. This would have helped during line traversal, since there were a different number of cells in the rows and the columns. This was done by creating some dummy threads that remained idle during the kernel, participating only in synchronization. As can be seen from the data, this significantly reduced computation time on the simpler boards, suggesting that a large part of the computation time on the simple solving is a result of divergent execution.

Reducing synchronization points within each kernel affected running time significantly. The algorithm used involves solving from the perspective of the run constraints and then from the perspective of the blocks already on board. Removing the synchronization between these two phases, which allows them to run concurrently, as well as the synchronization between the column and row phases, provided a 10% reduction in computation time on #1739. This was despite the possibility of requiring more iterations to arrive at the solved state or a dead end.

3.3.3 Puzzle availability

Because of the nature of human-friendly nonograms, even the hardest nonograms from webpbn.com were solved in less than 1 second on both the sequential

and parallel solvers. This lack of large puzzles somewhat hindered the testing of the parallel solver

3.4 Potential improvements

One possible improvement would be to use a sparse matrix data structure for the hypothetical board, storing only changed cells. This may help since there will not be a need to duplicate the master board to work on it. However, this would likely not work well on a GPU since there would be a lot of synchronization required with such a data structure.

It is also possible that a different, more expensive heuristic not suited for sequential execution could work better on a GPU to reduce the number of guesses required. However, since execution time is already so short, it is unlikely to be significantly faster than the current system with the current heuristic.

Finally, using a CPU with a large number of cores such as a Xeon Phi which could handle divergent execution better might increase performance. Nonetheless, data sharing especially for the board will be slower on such a device, which would affect the speedup that could be obtained. Thread creation overhead would likely also be an issue due to the already short computation time of the solver.

3.5 Theoretical limits

As theorized in the approach section and confirmed by the measured results, being able to parallelize the simple line solvers did not improve performance enough to make up for the overhead added by CUDA. In part this was because the puzzles were too small. Increasing the work done by each individual line solver would increase the performance of the parallel implementation relative to the sequential implementation. However such a puzzle might also exceed the resources available to most GPUs.

Consider a puzzle with 400 rows and 400 columns. It would require 16x the memory as the largest puzzles that were able to be found for this benchmark. However, such a puzzle would only have 800 total line solvers, merely 4x as many. In general it can be noted that for a square puzzle with length n along each dimension, the resource requirements grow with $O(n^2)$ while the benefit from parallelizing the line solvers only grows with $O(n)$. While such a puzzle might not exceed the resources of the entire GPU, it would almost certainly be too large for a single thread block.

Additionally, though it was not included in the timing of the runs of the parallel solver, the resource requirements of the solver include overhead of transferring data to the GPU.

4 Distribution of work

Roughly 2/3 of the work was performed by Benjamin, who implemented the majority of the solver. 1/3 was performed by Eric, who wrote the majority of the proposal and the reports.