Ultimate Tic-Tac-Toe

15-418 Final Project
Srishti Srivastava and Summer Kitahara
scrabble-bot.weebly.com

1 Abstract

We implemented and parallelized an algorithm that plays N-dimensional "ultimate" or "meta" tic-tac-toe against itself (NxN tic-tac-toe boards of size NxN running at the same time). Our algorithm is based on alpha-beta pruning, which we tried parallelizing in two different ways using OpenMP. While we achieved modest speedup, we discovered that parallelizing such a heavily recursive algorithm is probably not worth the effort.

2 Background

2.1 Terminology

- Meta-board: the board overall, which consists of NxN "mini-boards" that each are a tic-tac-toe game of size NxN.
- Mini-board: one of the smaller tic-tac-toe boards inside the meta-board. Winning a mini-board is equivalent to placing your tile in that space on the meta-board.

2.2 Rules

"Ultimate" or "meta" tic-tac-toe is a variation of tic-tac-toe where there are NxN boards of size NxN running simultaneously. As shown in Figure 1, if player X plays in space (i, j) in the grid at (k, l) on the "meta-board," that means that player O must play in the grid located at (i, j) on the meta-board. If O is forced to play in a board that has already been won or tied, O can instead play anywhere on the meta-board. Winning a given small grid serves as placing a tile in the larger NxN meta-grid. The goal is to win the overall meta-grid. The game ends when someone wins or there are no legal moves to be made (i.e., every mini-board has been won or tied).

2.3 Randomization

In our version of the rules, player O always goes first in the following manner: a grid on the meta board is randomly selected and then a row and column within that board is randomly selected to place the first "O". Because of this, the game has a different outcome every time, and the number of moves per game varies up to 30 moves for a 3x3 board and 150 moves for a 5x5 board. This also gives a very slight advantage to player X, who has one more optimized move than player O has.

2.4 Alpha-Beta Pruning

Alpha-beta pruning is a more efficient optimization of the naive

minimax algorithm. Minimax is used to compute the optimal move in two-player adversarial games by assigning a score to the current game state using a heuristic, and then hypothetically making every possible move recursively.

At each level, minimax alternates between the optimal move for the opponent and the least optimal move for the player. For each hypothetical move, alpha is updated to be the maximum of itself and the current move's heuristic value if it is the maximizing player's turn. Otherwise, it is the minimizing player's turn, so beta is updated to be the minimum of itself and the current move's heuristic value. Hypothetically making these moves generates a tree of possible paths the game could take. Alpha-beta pruning shrinks the problem size by immediately discarding suboptimal moves (for example, moves that would result in the player losing,) and therefore prunes entire subtrees.

2.5 Heuristic

To quantify the "goodness" of a particular board state for a particular player, we modified "heur2" described here. At a high level, the heuristic assigns a hierarchy of points on the meta-board for

- Winning the center board
- Winning a corner board
- Making a move in the center board
- Winning any board
- Winning any combination of N-1 boards in a row.

In addition to this score, the heuristic assigns a hierarchy of points on each mini-board for

- Placing a tile in the center
- Placing any combination of N-1 tiles in a row.

The heuristic is symmetric, so the opponent receives an equivalent negative score for each of these features.

3 Approach

We used OpenMP to parallelize because we wanted to parallelize a recursive algorithm. Two intuitive approaches to this problem are iterating over the children in parallel, or creating a pool of shared work from each child. We found that OpenMP has good support for both these approaches in the form of parallel for-loops and task abstraction. Additionally, since this recursive problem space will inherently increase exponentially, we decided to experiment with keeping portions of the algorithm sequential to avoid the overhead of managing exponentially growing parallel processes.

3.1 Sequential Alpha-Beta Implementation

In our implementation of alpha-beta pruning, each recursive step computes the minimax values for the two players. At each level of recursion, it is a different player's turn. This means that every other call to the function tries to maximize the score for the current player and all of the other turns are trying to minimize the score.

First, we collected all of the children, which are empty squares within the current mini-board. Then, we placed a hypothetical tile at one child at a time, evaluated the board with the heuristic function, and recursed on this updated board. After the maximum depth, which we specified to be 4, is reached, the algorithm returns the move that maximizes the maximizing player's score and minimizes the minimizing player's score. We found that increasing the max depth beyond 4, and therefore creating exponentially more child nodes, made the algorithm far too slow (parallelized or not).

The computationally expensive aspect of the alpha-beta algorithm is iterating through the children and making recursive calls on each one. The results of the child nodes do not affect one another until they return and are compared to one another, making the computation of the child nodes our target for parallelization.

3.2 "Meta-move" Implementation

As the rules for "meta" tic-tac-toe state, if the player is sent to a mini-board where no legal moves can be made, they can play in any other available mini-board. We called this a "meta-move." In order to account for this possibility, we divide heuristic calculation into meta-board components and mini-board components. Choosing the "meta-move" involves computing the current meta-board component of the heuristic, iterating over all incomplete mini-boards on the meta-board and then executing alpha-beta pruning on the each of these mini-boards (using the mini-board component of the heuristic). Then the function returns the optimal mini-board for the current player to choose as well as the best move within the selected mini-board.

3.3 Input Size and Number of Threads

We decided to only run our program on N = 3 and N = 5. Any board sizes larger than this took too long to execute. Also, it is best if N is an odd number because the heuristic function gives weight to making moves in the center square, which could not be achieved if N were even.

To best understand the progression of speedup, we chose to examine the results with 1, 4, 8, 16, 32, and 64 threads for all cases, both when N = 3 and when N = 5. Especially for the N = 3 case, we knew that having more threads than possible spaces on the meta-board would only lead to increased overhead costs with no real benefits. However, for some of the N = 5 experiments, we tested with 128 or 236 threads when speedup was on an upward trend, to see if increasing the level of parallelism would improve our results.

3.4 Tasks Approach to Parallelization

A standard approach to parallelizing recursive calls is to create a work queue for parallel threads. We did this using OpenMP's tasks structure, `#pragma omp task`. For each empty space, we defined the work of hypothetically

making a move, computing the updated heuristic, and recursively executing the alpha-beta pruning algorithm as a task. These tasks can either be executed immediately or be implicitly added to a pool for idle threads to steal. Since alpha-beta pruning is a recursive algorithm, the number of queued tasks per level of the alpha-beta tree grows exponentially. Additionally, this creates a long set of parent tasks that do nothing but wait for child tasks to return before the can do anything. Instead of this, we performed alpha-beta pruning sequentially after a certain depth is reached in the game tree so that a greater proportion of threads have meaningful work to do, and any idle threads can find work in the task pool.

3.5 Parallel For-Loop Approach to Parallelization

The "meta-move" routine has one for-loop that iterates over all incomplete mini-boards on the meta-board and `num_threads` threads are spawned. This for-loop is parallelized using `#pragma omp parallel for`. In addition, each recursive step of the alpha-beta pruning function has two nested for-loops, which each have N steps (to iterate over the NxN grid.) Each of these for-loops is also parallelized using `#pragma omp parallel for` and `num_threads` threads are spawned. The number of threads spawned grows exponentially. Therefore, we additionally experimented with limiting the number of spawned threads by running the recursive calls that are lower than a certain depth in the tree sequentially. Therefore, parallel threads spawned by the upper-level calls had substantial work in executing these sequential calls, instead of just spawning children and waiting for them to return.

4 Results

In the figures in the appendix, raw data is summarized in graphs showing the number of moves vs. the total computation time to run a complete game (until someone wins, or there is a tie and there are no more legal moves). Each thread count has its own color to show where the timings compared to the number of moves "cluster". The average speedup is calculated using the average time taken by each thread count across 5 trials.

4.1 "Partial task queue": Tasks queued until depth < 2

For a 3x3 board, the speedup graph in Figure 5 shows a peak when 8 threads are used and for a 5x5 board, there is a peak in Figure 3 when 16 threads are used. This is because 8 threads is the closest number of threads to the board size, which is 9. Therefore, there are not too many threads such that many of them are idle and there is a lot of overhead and not too few threads such that each thread's workload becomes heavy because it did not queue any tasks past a certain depth in the game tree. The same goes for 16 threads, which is the closest number of threads to the 5x5 board size.

4.2 “Partial for-loop”: For-loops parallelized until depth < 2

Once the recursion depth becomes less than 2, which is at the bottom and second-to-last level of the game tree, we decided the execute the remaining alpha-beta pruning sequentially. When N = 5 in the parallel-for approach, as seen in Figure 7, the speedup dips before rising again when there are 4 threads. This is because there are 4 threads spawned on the outer for-loop and then the inner for-loop. Since 4 is less than 5 and 16 is less than 25, the 16 spawned threads have to stall at the implicit wait at the end of the for loop before finishing the rest of the iterations of the for loops.

4.3 “Complete for-loop”: All for-loops are parallelized

As in Section 4.2, the speedup for N=5 as seen in Figure 11 also dips before rising again when there are 4 threads. However, the dip is less severe because each thread has less work, and therefore its parent threads have to wait for less time. When every for-loop is parallelized for every level of the game tree, each thread does not have the burden of also talking on sequential work for the children in its subtree. However, this approach does not have as good of speedup as the approach discussed in Section 4.2 because the overhead of spawning more threads is larger than the actual work each thread accomplishes.

4.4 General Discussion and Comparison

As shown in Figure 3, our algorithm achieved a max speedup of 2.78x on a 5x5 board using Partial Task Queue approach. This is because the work pool allowed for task stealing amongst threads, which decreased the amount of idling per thread. In contrast, in the parallel for-loops approach, there is an implicit wait at the end of each of the nested for loops, and threads that go idle can’t pick up more work until after the wait.

After completing this project, we learned that parallelizing recursive functions may not be worth the trouble. Since the number of threads spawned grows exponentially, the overhead of distributing the work and managing all of these threads starts to overpower the parallelism. We tried to compensate for this by running the algorithm sequentially and not creating any more tasks after a fixed number of recurses.

4.4.1 Limitations on Speedup

Our speedup was limited by inherent overhead work. This includes finding the “children,” or blank tiles on a board, at each step, calculating heuristic values of game states, and dividing work up amongst threads. In addition, the maximum problem size was a board of 625 tiles with a max recursive depth of 4. Thus, our speedup was also limited because the ratio of problem size to number of threads was never very large. And finally, parallelism is inherently limited in recursive algorithms because at each

level, parent nodes must wait for their child nodes to return before they can do meaningful work.

4.4.2 CPU v GPU

In the future, we would experiment switching from CPU to GPU because GPUs have better throughput and smaller caches than CPUs do. Our algorithm does not require a lot of memory, so perhaps we could gain faster runtimes on a GPU.

5 References

http://www.cs.huji.ac.il/~ai/projects/2013/UlitmateTic-Tac-Toe/

https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe

http://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf

https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

https://docs.oracle.com/cd/E19205-01/820-7883/6nj43o69j/index.html

6 Total Distribution of Credit

Summer Kitahara - 51%
Srishti Srivastava - 49%

Appendix A: Graphics

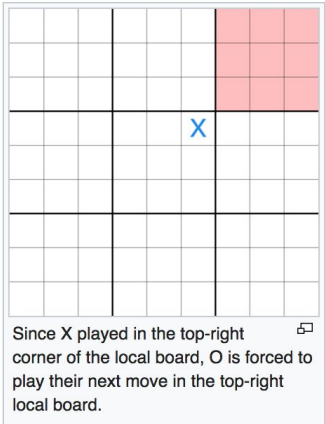

Figure 1

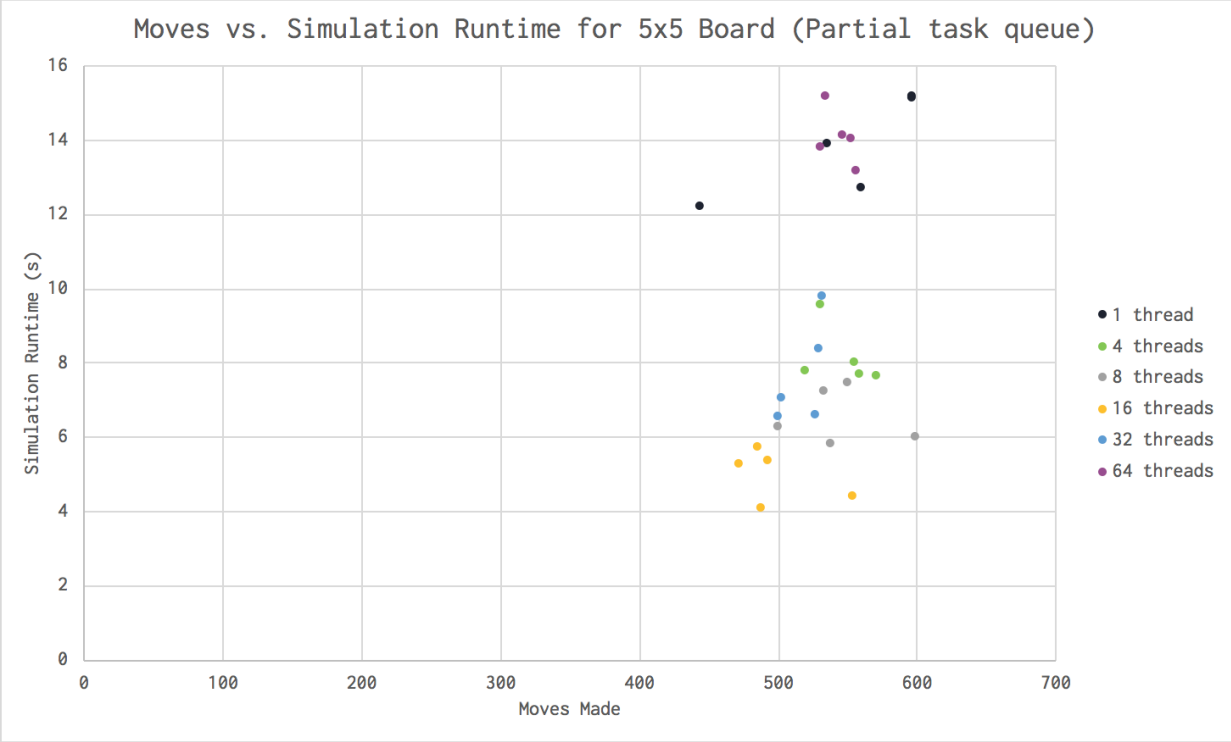

Figure 2

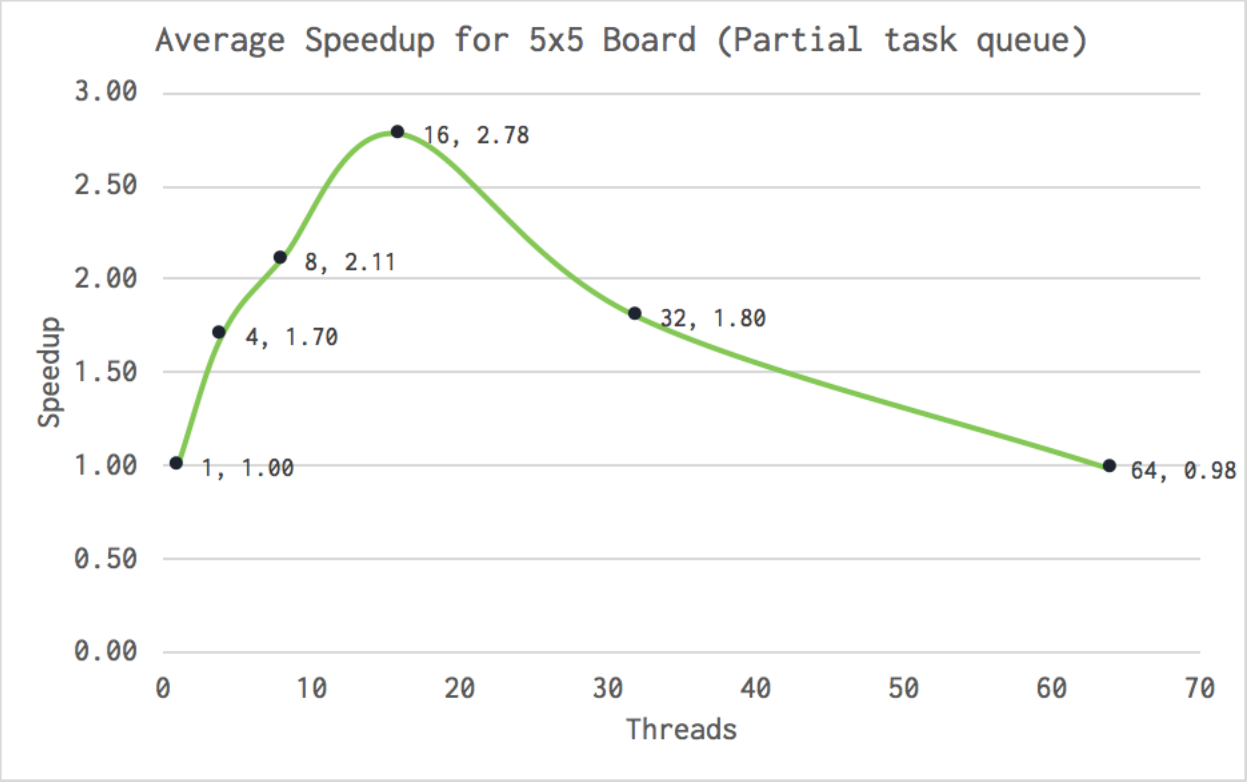

Figure 3

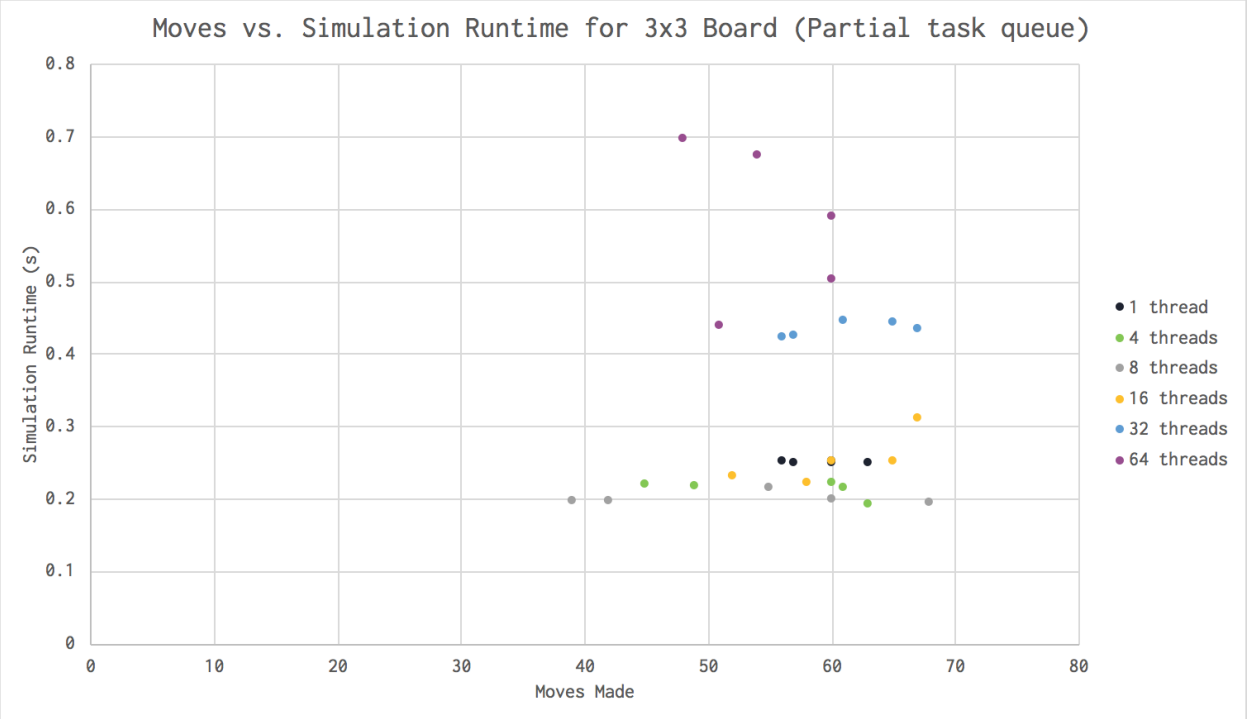

Figure 4

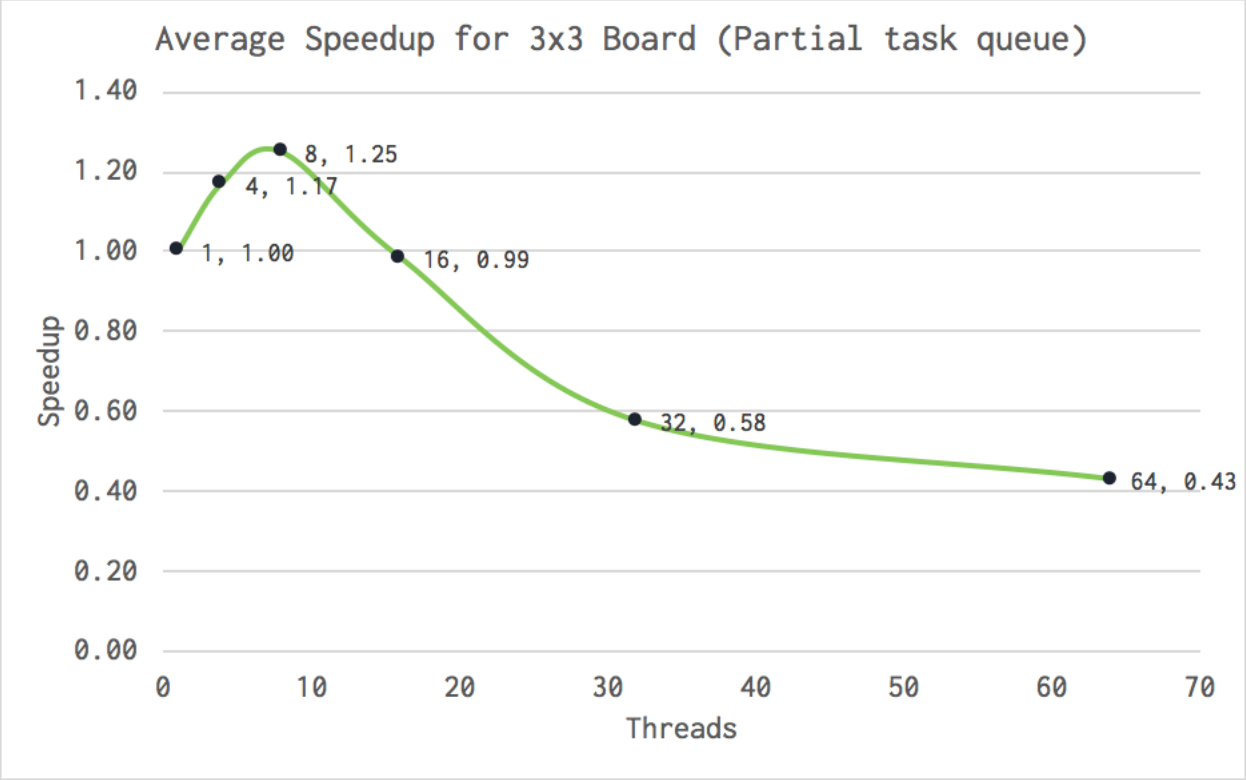

Figure 5

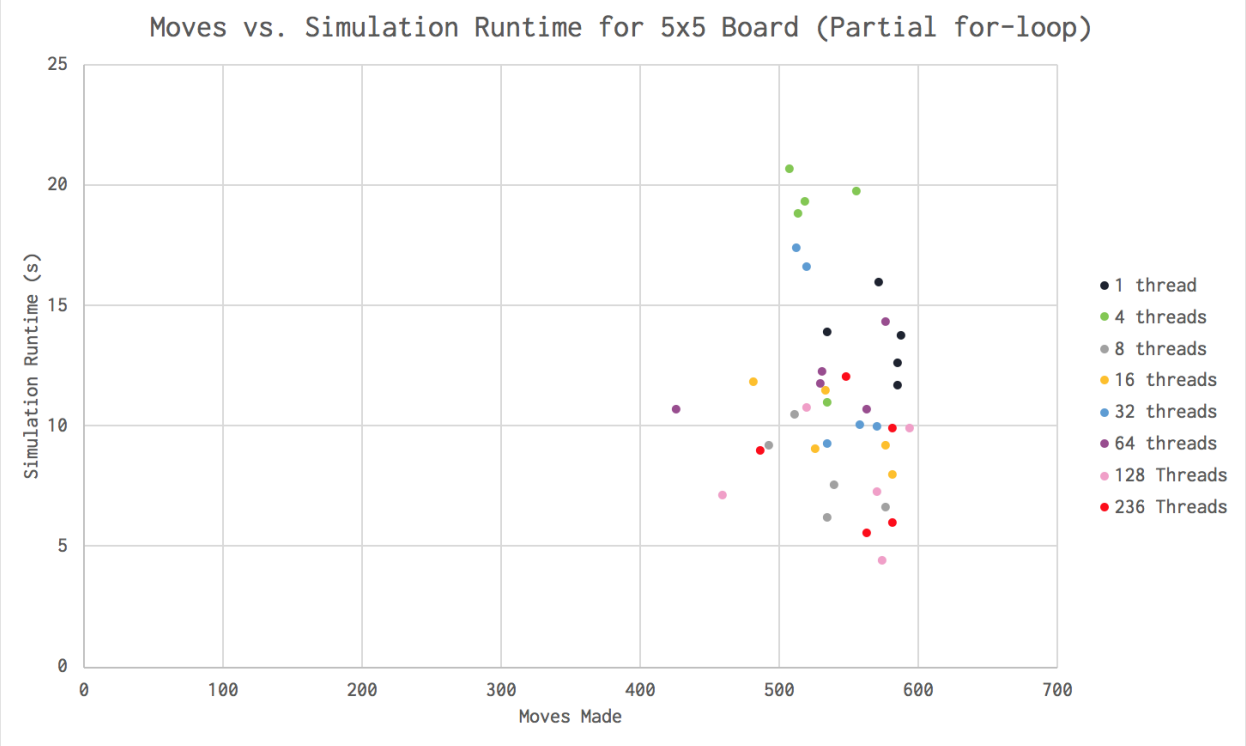

Figure 6

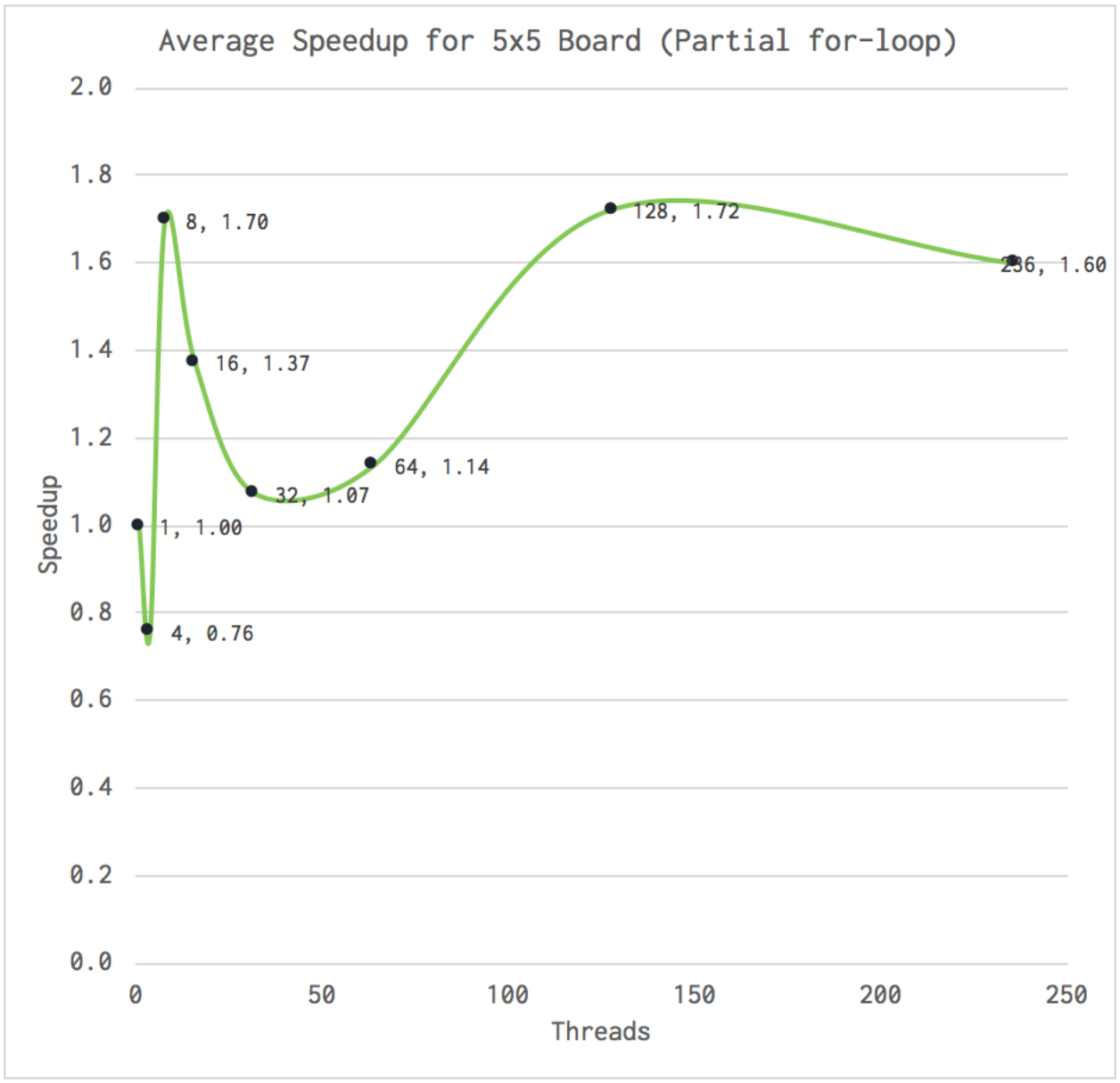
Average Speedup for 5x5 Board (Partial for-loop)
Speedup
Threads
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
1, 1.00
4, 0.76
8, 1.70
16, 1.37
32, 1.07
64, 1.14
128, 1.72
256, 1.60

Figure 7

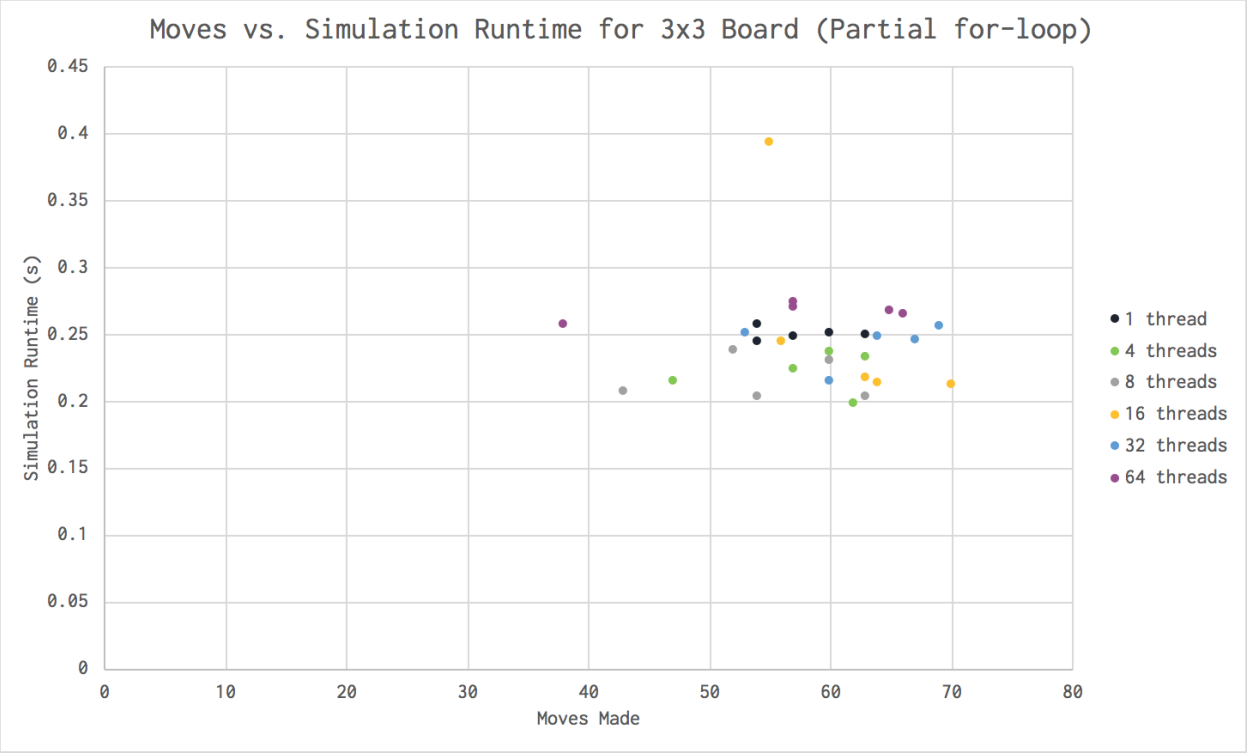

Figure 8

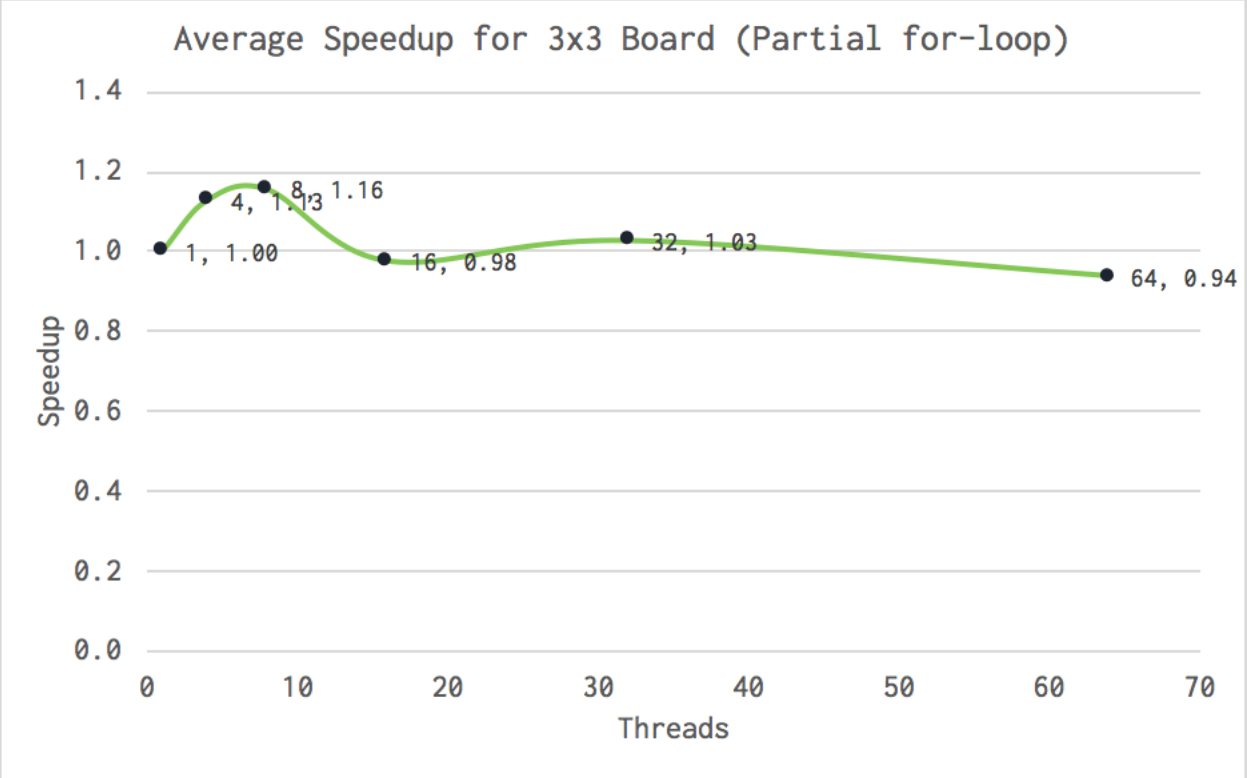

Figure 9

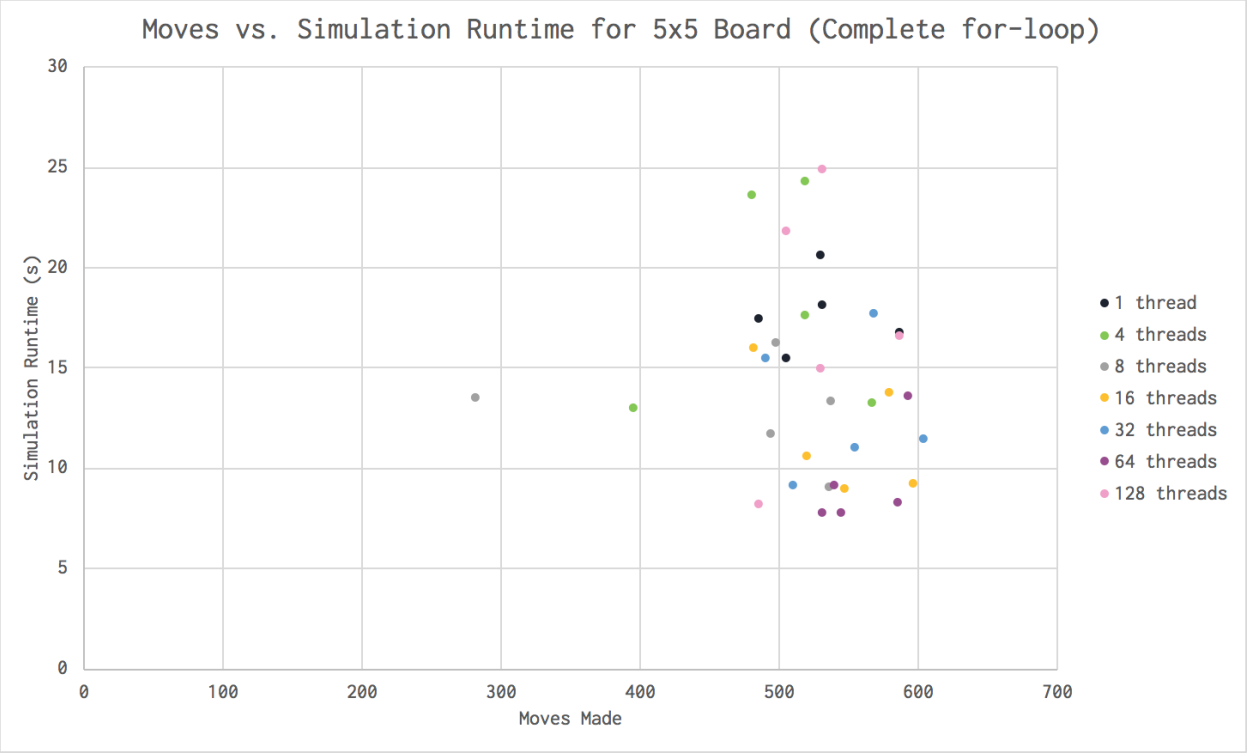

Figure 10

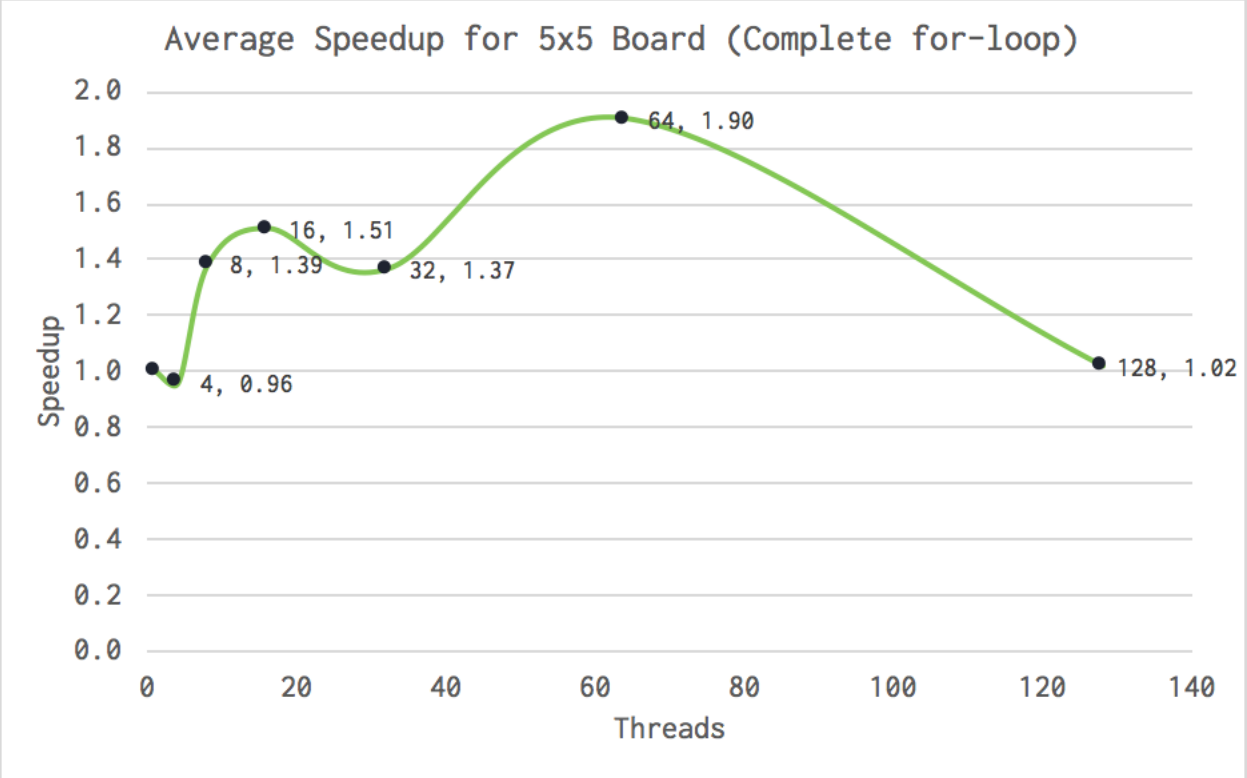

Figure 11

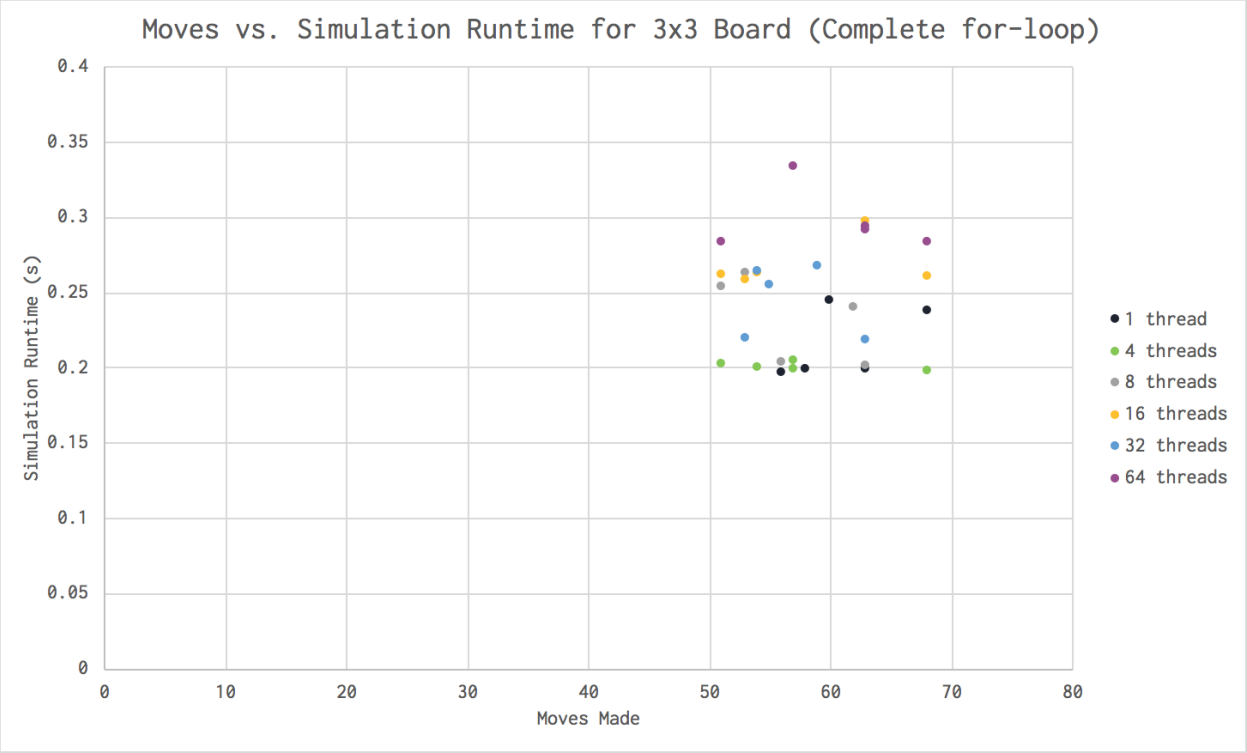

Figure 12

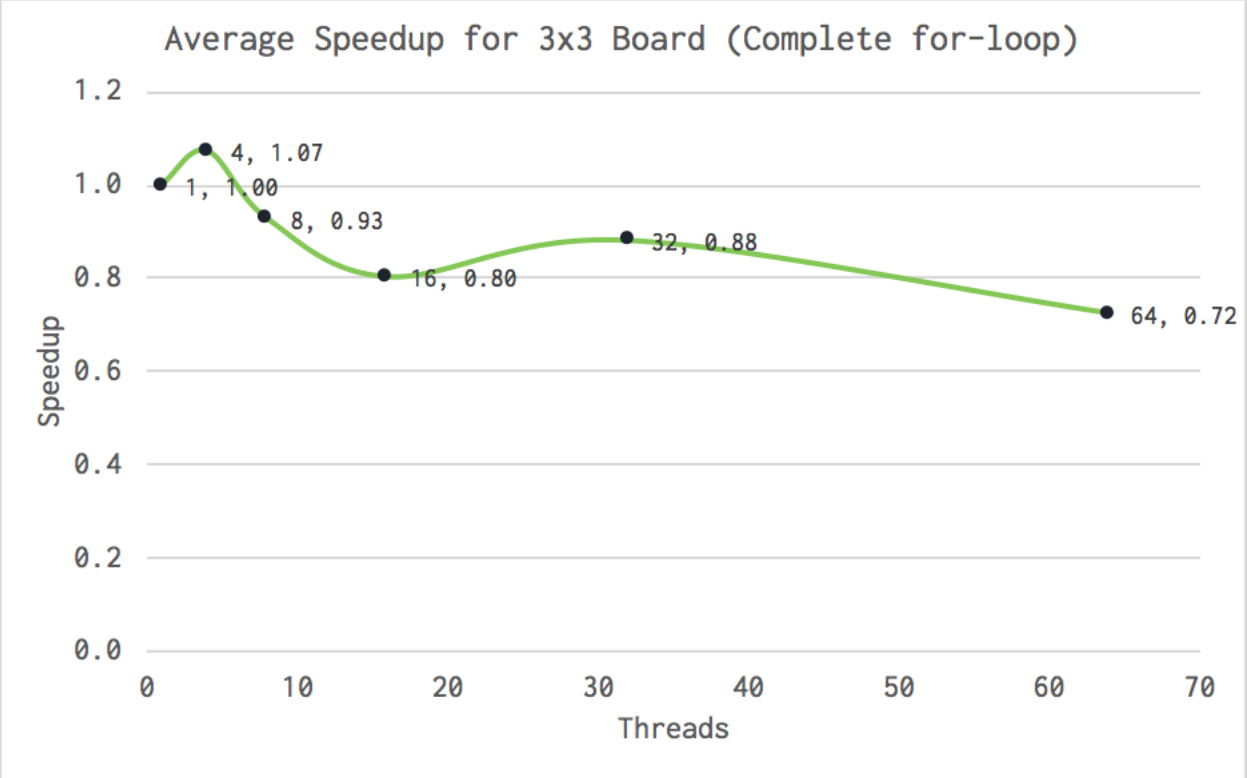

Figure 13