Solving Sudoku using Particle Swarm Optimization on CUDA

Jason Monk, Kevin Hanselman, Robert King, Raymond Flagg
Yifeng Zhu PhD., Bruce Segee PhD.
Department of Electrical and Computer Engineering
101 Barrows Hall
University of Maine
Orono, ME, 04469

Abstract—Sudoku is a popular puzzle utilizing 81 squares in a 9x9 grid consisting of nine 3x3 boxes. The digits 1-9 can each appear only once in a given row, column, or box. This paper describes the implementation of Particle Swarm Optimization (PSO) to solve sudoku puzzles using GPU processing. This PSO uses our open-source PSO framework that takes advantage of CUDA-enabled GPUs. Although each row contains nine digits, permutations of nine digits can be represented as eight “picks”. To find a solution each of the nine rows was treated as a permutation. This reduced the problem dimensionality from 81 to 72. With suitable parameters the algorithm was able to solve multiple sudoku puzzles. This paper describes the implementation of the algorithm, the fitness function used, and the effects of variation on PSO parameters. The original PSO framework and the Sudoku code described in this paper are available online.

I. INTRODUCTION

An open source framework for implementing Multi-swarm Particle Swarm Optimization on GPUs using CUDA was developed and discussed in [1]. The framework was previously demonstrated on a problem to optimize the parameters of a PID controller. This was a relatively well behaved problem in a three dimensional space. It was shown that by utilizing GPU processing, the problem could be solved many times faster than was possible using the CPU.

In the previous paper it was claimed that the framework could easily be extended to other problems and higher dimensional spaces. In this paper we utilize the framework to find solutions to Sudoku puzzles. These represent a class of problems in a very high dimensional space. Furthermore, the space has large number of local minima that occur at large distances from the global minimum.

We believe that the solution of Sudoku puzzles represents a very significant optimization problem. In this paper we show that 1) the open source Multi-Swarm Particle Swarm Optimization Framework that we have previously developed is in deed general enough to extend to this problem, 2) that the dimensionality of the problem can be reduced using permutations, 3) that the use of GPUs is crucial for reducing run times from multiple days to hours, 4) that PSO can indeed solve Sudoku puzzles, and 5) that the choice of PSO parameters has a huge impact on the time to find a solution.

II. SUDOKU

Sudoku is a logic puzzle that has been extremely popular in the US since about 2005 and is based around using rules and logic to determine where numbers belong. This section describes basics of the sudoku puzzle as well as its origination.

A. Origin

Sudoku is often thought to be of Japanese origin, but this is not actually true. [2] Modern Sudoku first appeared in American papers in the 1980s, but did not become popular in the US until 2005. Although the puzzle we know today only dates back around 30 years, similar puzzles originated in the late 19th century in France, where puzzles with very similar solutions to sudoku originated. [3]

B. Rules

The rules to a sudoku are to fill a nine by nine grid with nine three by three sub-boxes such that only one instance of each of the digits one through nine is contained in each column, row or box. This means in any given column or row there should no repeated numbers and no numbers other than one through nine.

C. Related Work

This was not the first time that Sudoku had collided with biologically inspired algorithms. Sudoku puzzles have been solved by both GA and GPSO in the past. This, however, is the first time using a more generic PSO and adjusting the fitness function accordingly.

Sudoku puzzles were solved by Mantere and Koljonen 2007 [2]. In the same year, Sudoku puzzles were also solved by Moraglio et al. [4], where they introduced combinatorial-based PSO algorithms (GPSO), these spaces were similar to the pick space used in this paper. Sudoku puzzles were confirmed to be solvable using GPSO by Jilg and Carter 2009 [5].
III. PARTICLE SWARM OPTIMIZATION

PSO was originally developed by Eberhart and Kennedy [6] in 1995. It is heavily biologically inspired and it mimics behaviors that can be seen often in various types of flocking or swarming animals. It is not a complex algorithm and performs very well in continuous spaces that do not have analytical solutions. PSO is an iterative algorithm in that it moves each particle, calculates fitness of its location and then repeats the process.

A. Inspiration

Particle Swarm Optimization was inspired by animals in nature. Many different types of animals travel in groups, and often they benefit from the knowledge of each other. In a school of fish, each individual can benefit from the others knowledge regarding food and predators. Flocks of birds can cover larger areas by spreading out, and informing the flock of any food found. PSO mimics this in that each particle has knowledge about the fitness (or happiness) of itself and of other particles in the swarm, and tends to move toward regions of better fitness.

B. Algorithm

As mentioned previously, particle swarm optimization is based on having many particles, or candidate solutions, moving around an error/solution space. This implementation considers the best particle the one with the lowest fitness value; however it would work just as well looking for particles with larger fitness values.

Each of the particles tries to move to a solution that is better than its current one. To do this, a particle moves in the direction of the best location of the swarm and the best location it has found so far. The particle does a random weighting of each so it will be heading towards both to some degree. The following is the equation that describes the movement towards the particle’s best location as well as the swarm’s best particle’s location.

\[ \text{mov} = \text{rand}()p_{\text{weight}}(p_{\text{best}}p) + \text{rand}()s_{\text{weight}}(s_{\text{best}}p) \] (1)

Where \( \text{mov} \) is a vector in parameter space representing a movement, \( p_{\text{weight}} \) and \( s_{\text{weight}} \) are scalars chosen by the programmer, \( p_{\text{best}} \) is a position in parameter space representing the location where a particle has been the happiest, \( s_{\text{best}} \) is a position in parameter space representing the location of the happiest particle in the swarm, and \( p \) is the current particle location. \( \text{rand}() \) is a randomly selected floating point number between 0 and 1.

Since \( \text{mov} \) is a vector in parameter space it generalizes to any number of dimensions depending on the parameter space. In the case of no momentum this vector is calculated and then added to the location each iteration.

The algorithm often performs better when each particle’s velocity has some momentum associated with it. [7] In the framework used, the velocity was implemented with momentum, such that the new movement was composed of 90% of its old movement plus the new movement calculated above. This helps prevent particles from getting stuck in any single location, such as local minima. A velocity of the particle is calculated and saved each iteration using the following equation.

\[ \text{vel}_{\text{new}} = .9\text{vel}_{\text{previous}} + \text{mov} \] (2)

Where \( \text{vel}_{\text{new}} \) is a vector representing the velocity or the amount that the particle will move this iteration, \( \text{vel}_{\text{previous}} \) is the velocity from the previous iteration, and \( \text{mov} \) is a vector calculated above in equation 1. Once the velocity is calculated it is added to the particles location every iteration.

C. Related Work

Our PSO framework was not the first one to reach the GPU scene; however it was the first with the goal of making the PSO framework more accessible. Zhou and Tan 2010 [8] implemented a PSO algorithm using GPUs. Rather than using multiple swarms it used a triggered mutation system on top of a standard particle swarm optimization. They were able to achieve a speedup of 25x with this system.

An asynchronous implementation of PSO was created by Mussi et al. 2011 [9]. This allowed each particle to run iterations at its own rate (which was very fast). However this was limited by the fact that only one particle was allowed per block, and the maximum number of blocks that can run in parallel limits the swarm size.

Vanneschi et al. 2010 [10] tested a multi-swarm system where the best particles from one swarm were passed to the next swarm to replace the worst particles, this was done in a ring setup of several swarms. They exchanged this information every 10 steps. This was implemented again by Solomon et al. 2011 [11].

IV. PARALLEL PARTICLE SWARM OPTIMIZATION FRAMEWORK

The Parallel Particle Swarm Optimization Framework was an implementation of the Particle Swarm Optimization algorithm that would use CUDA and be flexible to a number of different problems. The framework was designed so that a programmer with knowledge of C, but minimal knowledge of CUDA could modify the program to solve a large range of problems. In this case the framework was modified to solve Sudoku puzzles.

A. Multi-Swarm

Since each swarm is at some point in time entirely held in CUDA shared memory, the size of each swarm is limited. The following equation defines how many particles can be in a single swarm based on the dimensionality of the problem. [1]

\[ \text{Max\#ofPart} = \frac{16,384}{8 + 12 \ast \text{DIM}} \] (3)
Since in our problem each particle will be moving within a 72 dimensional space, 72 can be substituted, giving the actual maximum number of particles.

\[ \text{Max\#ofPart} = 18 \quad (4) \]

Because of this limitation the framework allows for multiple swarms to be run in parallel. To allow the swarms to cooperate in some way, particles are swapped every 1000 steps. Each time a swap occurs there is a 1% chance that any 2 particles will be swapped.

**B. Modifications to Code**

The only programming required by the framework are generally modifying PSO parameters, such as weights, as well as rewriting the fitness function. The first problem tested was tuning of PID Controller parameters. While this problem was good for its computational intensity, it had a very small memory footprint. When using this framework to implement a sudoku solver a bit more memory was required.

The added dimensionality of this problem will still be held within the particles individual location; however the information specifying constraints to the puzzle is common to all particles. To make this quickly accessible to all threads it was stored in constant memory on the GPU. This memory is cached for all threads to make it easily accessible. Both the "pick" space and the solution space constraints were stored in this constant memory, which will be described in section IV-G.

**C. Sudoku Fitness Development**

For the PSO framework being used a fitness function had to be developed. This fitness function needed to be a function that could determine if one solution was better than another and by how much. It could specify how good a solution is by filling in a floating point value inside the particle structure when the fitness calculation function was called. A better solution is interpreted to be one that has a lower fitness value than another.

**D. Pick Space**

It was mentioned previously that a 72-dimensional space was being used to solve the Sudokus. This Sudoku solver is using a row based solution system, where each row is viewed as a permutation. Each row can contain the digits one through nine in any order, but each digit can only occur once.

This can also be interpreted as a sequence of selections from an ordered set of the numbers one through nine. It would start with an ordered set and a empty row as shown in figure IV-D.

Next there are a set of "picks" that select the order in which the numbers will appear in the row. If the first "pick" is assumed to be 5, the 5 would be moved from the available set to the row, this is shown in figure IV-D.

- Available: [1 2 3 4 6 7 8 9]
- Row: [5]

Fig. 2. One Pick Complete

If the next pick is also 5 then the 6 is now moved from the available set to the row set, finally shown in figure IV-D.

- Available: [1 2 3 4 7 8 9]
- Row: [5 6]

Fig. 3. Two Picks Complete

This process continues until the row set is filled and the available set is empty. This will only take 8 picks as once the last pick is reached, there is only one number available.

It can be observed that in this "pick" space the size of each dimension decreases with each pick, starting with 9 and decreasing down to the known pick with a size of 1. In this implementation if a pick was outside the space then the nearest number would be chosen, one if it is too low and the max of the dimension if it is too high. When these picks are extended to be 8 dimensions for each of the 9 rows the problem has a 72-dimensional search space.

This "pick" space turned out to be a good intuitive space for searching Sudokus because a movement in any dimension by one unit will create a swap of two numbers. The goal of the fitness function is to make it so puzzles similar to the solution will be only a series of swaps away from the actual solution.

**E. Fitness Function Basis**

The first implementation was the most intuitive implementation of the fitness function that could be found. The first system was very generic in that location in the puzzle did not affect fitness at all. It would first generate the candidate solution based on the location of the particle, then it would overwrite any constraint in the puzzle. Once this was done it would have an array representing an attempted solution to the puzzle as shown below in figure IV-E. The red numbers show constraints that have taken the place of whatever value was chosen based on particle location.

- Available: [1 2 3 4 6 7 8 9]
- Row: [ ]

To calculate the fitness each of the duplicates were counted. Since any number should occur only once in each row, column or box, any duplicate can be perceived as a problem in the puzzle and therefore the more duplicates the less fit a solution is. So every column, row, and box is scanned for duplicates and the fitness is set to the sum...
of every duplicate. The missing numbers could have been analyzed as well; however these are actually equal to the number of duplicates, as for every duplicate that exists there is one number that was overwritten and is missing.

This fitness function performed relatively well overall, it minimized to puzzles that had very few duplicates. The problem that arose was that sometimes a puzzle with few duplicates could be far from the solution. The solution space seemed to be littered with local minima for most problems.

F. Modified Fitness Function

To increase the effectiveness of the fitness function, several aspects of the sudoku construction were observed. The first thing that lead to modifying the fitness function was that there were a large number of candidate solutions that had the same fitness value. There needed to be some way to determine if one of the solutions was better than another when they had drastically different structures.

The first modification was to make boxes to the left count more than boxes to the right. The reasoning for weighting more importance on left sided boxes is that the ”pick” space chooses from the left to the right. Weighting the left side of the puzzle more heavily encourages the algorithm to fix the first dimensions of the picks in each row before moving onto the smaller dimensions on the right.

The second modification was to add a component to the fitness for having an incorrect constraint. Although the constraint boxes are overwritten to find the solution, having a wrong constraint can make it more difficult to get the correct permutation of the row. This fitness addition carried the same left-weightedness as stated in the first modification.

The final modification was to attempt to keep particles within the search space. Although being outside the search space does not cause any problems with the algorithm, it is known the actual solution will reside at a location within the range of the pick space. Because of this the fitness was increased for each dimension that was outside the bounds for that given pick.

G. Pick Space Constraints

Although the constraints push the problem in the correct direction through the fitness, there is some knowledge about the pick space locations of the true solution that can be known immediately. Figure IV-G shows only the constraints of a given puzzle that was solved, where the highlighted values are ones that have known pick constraints.

V. RESULTS

At first the PSO based Sudoku solver did not work at all. Most PSO parameters showed it unable to even come up with good attempts at solving the puzzle. However through trial and error a set of parameters giving good performance were found. Since the space was so complicated a low swarm best weight was used (0.05), a high local best weight was used (1.0), with a fairly normal momentum of 0.9.

These were the first parameters that were able to solve the Sudoku successfully. The region around these parameters was analyzed and as described later in this section these were very close to the most optimum parameters for finding solutions
Once a solution had been found, most of the tests were run with a max number of iterations set to 1.65 Million. These tests took a half hour each on the CUDA-based PSO framework used. These same tests when run using the computers CPU, rather than GPU took slightly over 24 hours, making the speedup approximately 48. It is noteworthy that the runs described in this paper utilized the GPU processors of a GTX 260 for approximately a week of total run time. These same runs would have taken approximately a year to run on a conventional computer.

A. Solved puzzles

The first puzzle solved was the puzzle shown in IV-G. It also solved the following puzzle in figure V-A.

\[
\begin{array}{cccccc}
5 & 3 & 8 & 6 & 4 & |\\
9 & 2 & 3 & 7 & 6 & 4 & 5|
\end{array}
\]

\[
\begin{array}{cccc}
9 & 2 & 1 & 2|
9 & 3 & 1 & 4|
8 & 7 & 6 & 4 & 2|
3 & 7 & 5 & 1 & 9 & 4 & 2|
4 & 3 & 8 & 9 & 7|
\end{array}
\]

B. Effect of parameters

It was difficult at first to find any PSO parameters that would solve the Sudoku puzzles given. Because of this the effect of the PSO parameters was studied. A series of tests, each with a maximum of 1.65 Million iterations, were run on various changes in the PSO parameters. Four random seed values were chosen, for each seed a specific set of starting particles exists. Each of the four seeds were used on every set of parameters tested.

The results showed that having the momentum at 0.9 was very much a perfect spot for solving sudoku. Changing the momentum in either direction seriously interfered with the ability to find a solution. Figure V-B shows the average ending fitness compared to the momentum.

There is an upward trend as both of the weights increase. This could suggest that the algorithm is simply taking more time and it will eventually find a solution, or it could suggest that if the weights increase too much it will be unsuccessful in finding a solution. In other isolated tests larger weights did not perform well and often would not converge or would converge in very high local minima not near the solution.

C. Random Addition

Even a small amount of testing showed that this was very clearly a very bumpy search space, full of local minima. This was shown in the last section by the heavy weighting upon particles local best and light weighting on swarm best locations. Despite this weighting some tests revealed it was possible for the algorithm to come very close to finding a solution without actually finding it.
The solution to this problem was a random addition. This had no biological basis, and the random numbers had such a small coefficient that it had virtually no affect on the running of the algorithm. However despite its small nature it did help in scenarios where the entire swarms were stuck at a single local minima. The random addition to all particles location would give them some momentum, allowing more searching of the space.

VI. Conclusions

This implementation successfully solved multiple Sudoku puzzles and has yet to be attempted on a puzzle that it did not eventually solve. The speedup of the CUDA-based PSO framework that was used was instrumental in the success of this project, as even to run 1.6 Million iterations on the GPU still took on the order of 30 Minutes to complete.

Due to the bumpy nature of the parameter space a much smaller swarm best location weight than usual must be used to solve the Sudoku puzzles using PSO. This problem also proved to be extremely sensitive in variation in the momentum coefficient.

VII. Acknowledgements

This work was partially funded by NSF grants EAR 1027809, OIA 0619430, DRL 0737583, CCF 0754951, EPS 0918018, EPS 0904155, and NIH grant R01 HL092926.

References