CMPE 297 HW#3

Due: Friday, Oct 21, 11:59pm
Total Score: /100

Instructor: Hyeran Jeon

Computer Engineering Department, San Jose State University

Homework is not a group work. Each student should submit his/her own homework solution.

In this homework, you will implement a parallel version of a brute-force algorithm to solve "Easy" Sudoku puzzles where one square's value can always be deduced each iteration until the entire puzzle is solved.

Sudoku: Sudoku is a number-placement puzzle where the objective is to fill a 9×9 grid with digits 1 through 9, such that each column, each row, and each 3×3 sub-grid (called cell) contains all of the digits from 1 to 9. Rule: Numbers from 1 to 9 should be in each row, column, and cell only once.

An example is illustrated below:

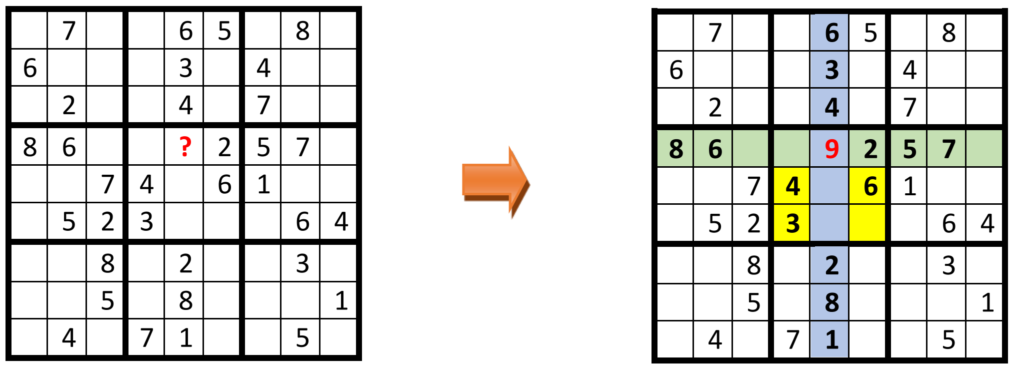

Figure 1 –An example Sudoku matrix.

Given a partially completed matrix (left), the number for (row=3,col=4) can be figured out. As row 3 already has {2, 5, 7, 8, 6}. Column 4 has {1, 2, 3, 4, 6, 8}, and the cell (1,1) (the 3x3 sub-matrix in the middle of the 9x9 matrix) already has {2, 3, 4, and 6}. Looking over all these numbers, the only one missing is 9. Hence (3,4) is set to 9.

Sudoku Solver: A simple algorithm to solve the Sudoku goes over each entry in the 9x9 grid, placing values and checking if any of the rules are violated. If not, the algorithm moves to the next and repeats, until a solution can be found. Although there are upwards of 10^{20} possible final grids, the exact time that it takes to solve a given problem depends on the initial values. A simplified version of the brute-force algorithm can be used for “very easy” starting grids that can be solved by progressively plugging in values where only one option for that square exists (i.e. an obvious solution). This algorithm assumes that in each iteration, there is at least one such obvious element.

```
while(unknown entries exist)
  for (row 1 to 9; row++){
    for(col 1 to 9; col++){
      find all the values that are valid for entry (row,col)
      if (num of valid values == 1)
        set (row, col) to the only possible valid value
} } }
```

Sequential C code: To write the algorithm, some data structures are needed to maintain the numbers that are not in each row, column, and cell. Two dimensional arrays **not_in_row[r][k]**, **not_in_col[k][c]**, **not_in_cell[l][k]** store 1 (true) and 0 (false) values to denote that the number k+1 has not been set in row r, col c and cell L, respectively. That is, not_in_row[3][7] = 1 denotes the row 3 does NOT have number 8 in that row (recall that because arrays start with index 0, entry 7 (k) corresponds to number 8 (k+1).

val[r][c] and **num_options[r][c]** store the value and the count of the candidate numbers for each entry satisfying the rule (i.e. num_options[r][c] = 1 means (r,c) has an obvious choice that can be deduced from the not_in_row, not_in_col, and not_in_cell arrays. Table 1 summarizes these data structures.

Data Structure	Description of Usage in the Code
val[9][9]	The value for entry (r,c) with 0 meaning unknown.
num_options[9][9]	The number of candidate numbers that each entry can have. For example, if (2, 3) can have five numbers, 1, 3, 5, 8, 9 for its value, num_options[2][3] = 5

not_in_cell[9][9]	The numbers not in each 3x3 sub-matrix(cell). For example, if inside of cell 2 are numbers 1, 2, 3, 4, and 9 (i.e. 5, 6, 7, and 8 are NOT in the cell).
	not_in_cell[2][0] = 0, not_in_cell[2][1] = 0, not_in_cell[2][2] = 0,
	not_in_cell[2][3] = 0, not_in_cell[2][4] = 1, not_in_cell[2][5] = 1,
	not_in_cell[2][6] = 1, not_in_cell[2][7] = 1, not_in_cell[2][8] = 0
	The cell numbering is illustrated in Figure 1**.**

not_in_row[9][9]	The numbers not in each row. For example, if row 2 has numbers 1, 2, 3, 4, and 9 (i.e. 5, 6, 7, and 8 are NOT in the row).
	not_in_row[2][0] = 0, not_in_row[2][1] = 0, not_in_row[2][2] = 0,
	not_in_row[2][3] = 0, not_in_row[2][4] = 1, not_in_row[2][5] = 1,
	not_in_row[2][6] = 1, not_in_row[2][7] = 1, not_in_row[2][8] = 0

not_in_col[9][9]	The numbers not in each column.

Table 1: Key data structures for the Sudoku solver.

Cells are numbered from left-to-right and top-to-bottom, as shown below:

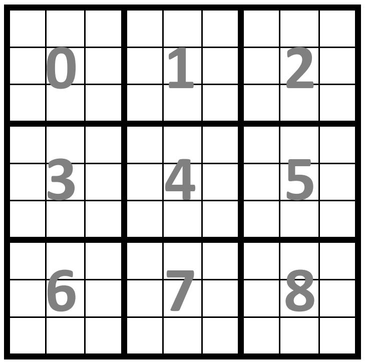

Figure 1 – The cell (3x3 sub-matrix) numbering.

We will place all of these arrays in a struct (like a class) called "**context**" that contains all the data our computation needs.

The main loop that checks and sets values in the grid is given below. Complete C code is downloadable from Canvas.

```
// Execution finishes when all the entries in the matrix have 1 value
while(unknown entries exist){
        for(int row = 0; row < 9; row++)  {
                for(int col = 0; col < 9; col++)  {
                        if(context.num_options[row][col] > 1)    {
                                // Find values that are options for the row, col
                                // & 3x3 cell that (row, col) belongs to.
                                int value = 0, temp;
                                context.num_options[row][col] = 0;
                                for(int k = 0; k < 9; k++) {
                                  temp = IS_OPTION(row,col,k);
                                  if(temp == 1)      {
                                        context.num_options[row][col]++;
                                        value = k;
                                  }
                                }

                                // If the above loop found only one value,
                                // set (row, col) to value
                                if(context.num_options[row][col] == 1)   {
                                  context.not_in_row[row][value] = 0;
                                  context.not_in_col[col][value] = 0;
                                  context.not_in_cell[(row)/3+((col)/3)*3][value] = 0;
                                  context.val[row][col] = value+1;
                                }
                        }
                }
        }
}
```

Requirements and guidelines:

The requirements/procedure steps are as follows:

a. Download the skeleton code from Canvas.

b. Complete the appropriate calls to cudaMemcpy to copy data to the GPU and then back once the kernel has finished.

c. Implement the function **__global__ void k_Sudoku(stContext *context)** in source file: cmpe297_hw3_sudoku.cu. Each thread should be responsible for a SINGLE entry in the Sudoku puzzle and iterate until its entry is known. The reference C code is also available in the Canvas.

d. Use the given **9x9 (81) threads** to parallelize the code.

e. Run your code and debug any issue until it is correct.

f. Once the code is functionally correct, optimize your code as much as possible. Measure your code execution time by using clock() API.

g. Comment your code enough for the TA/Instructor to understand your approach/intent to solving the problem.

h. Your code should be successfully compiled and run. Submit the code to Canvas before the deadline.

The skeleton code consists of a main function, a kernel function (the Sudoku solver), and three utility functions. The details are like below:

1) int main(int argc, char **argv) : Allocates memory on both CPU and GPU for the data structures and calls the kernel function.
2) __global__ void k_Sudoku(stContext *gpu_context): Finds the numbers for the blank entries of the given matrix. (the GPU kernel function)
3) void initialize_all() : Initializes the data structures according to the input matrix(const int input_sdk[9][9]).
4) void print_all(): Prints current matrix status. A 9x9 matrix with each entry having all the candidate numbers is printed to the standard output. The sample output is shown below.

Sample Output:

```
  0    *7*    0   |   0    *6*   *5*  |   0    *8*    0
 *6*    0     0   |   0    *3*    0   |  *4*    0     0
  0    *2*    0   |   0    *4*    0   |  *7*    0     0
---------------------------------------------------------------------
 *8*   *6*    0   |   0     0    *2*  |  *5*   *7*    0
  0     0    *7*  |  *4*    0    *6*  |  *1*    0     0
  0    *5*   *2*  |  *3*    0     0   |   0    *6*   *4*
---------------------------------------------------------------------
  0     0    *8*  |   0    *2*    0   |   0    *3*    0
  0     0    *5*  |   0    *8*    0   |   0     0    *1*
  0    *4*    0   |  *7*   *1*    0   |   0    *5*    0
 *4*   *7*   *1*  |  *9*   *6*   *5*  |  *3*   *8*   *2*
 *6*   *8*   *9*  |  *2*   *3*   *7*  |  *4*   *1*   *5*
 *5*   *2*   *3*  |  *8*   *4*   *1*  |  *7*   *9*   *6*
----------------------------------------------------------------------
 *8*   *6*   *4*  |  *1*   *9*   *2*  |  *5*   *7*   *3*
 *3*   *9*   *7*  |  *4*   *5*   *6*  |  *1*   *2*   *8*
 *1*   *5*   *2*  |  *3*   *7*   *8*  |  *9*   *6*   *4*
----------------------------------------------------------------------
 *9*   *1*   *8*  |  *5*   *2*   *4*  |  *6*   *3*   *7*
 *7*   *3*   *5*  |  *6*   *8*   *9*  |  *2*   *4*   *1*
 *2*   *4*   *6*  |  *7*   *1*   *3*  |  *8*   *5*   *9*

Processing Time : 6037.320137 (us)
```

The first 9 lines of the output is the initial status of the given matrix and the following 9 lines show the final result after the Sudoku solver execution. The entries that have 0 as their value in the initial status will be filled with numbers that are not in the row, column, and cell by the program. If the finalist is decided to an entry, the number is printed between two '*'s (i.e. *3*).