CS406 HW3

Baturalp Kabadayı

May 2023

1 Introduction

The goal of this homework was to design an implement a parallel algorithm in order to solve kakuro puzzles, using CUDA GPU programming library. You can check the GitHub repository for this homework to see all source code and extra samples of kakuro boards: kakuro_solver.

2 Algorithm Description

One approach to solve this problem would be a brute force search to try all possible boards. However, due to the nature of the problem, most of the possibilities can be eliminated if we use a correct search strategy.

First of all, a method is needed to identify which cell is included in which sum. For this purpose, a 2D array is created so as to map board cells to the sum indexes they are included in. A sum index is the place of a sum in sum arrays copied to GPU (For instance, d_sum_starts_x). This way, we only match cells and sums once and there is no need to find them again and again, which provides efficiency.

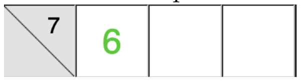
To be able to eliminate possibilities while searching for the solution, the problem is divided into $m * n$ sub-problems where m is the number of rows and n is the number of columns. Each sub-problem gets a list of boards as an input and produces new boards out of these boards. These sub-problems must run in order because of the fact that a sub-problems output becomes the next sub-problems input. Moreover, the first sub-problem's input is the empty board. After the last sub-problem is computed, the final output is the list of possible solutions, if there is any. Now the question is "What are these sub-problems?"

2.1 Description of a Sub-problem

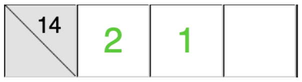
A sub-problem gets a list of uncompleted kakuro boards and tries to produce new boards by inserting all possible digits 1-9 to the cell where the previous boards are left. A new board (b_o) is created from (b_i) by inserting digit d if the sums which include the cell that d is inserted in are valid. A sum is valid if:

1) The sum does not include duplicates.
2) The numbers that are placed in the sum does not overflow.
3) The numbers that are placed in the sum does not underflow.

Here are examples of overflow and underflow:



This is an example of overflow. The remaining sum for last 2 cells is 1. Even though 1 and 2 (which are the minimum possible) are placed, their sum is 3, which makes this sum being valid impossible.



This is an example of underflow. The remaining sum for last cell is 11. Even though 9 (which is the maximum possible) is placed, the sum cannot add up to total 14.

2.2 Parallelization with CUDA

First of all, sub-problems cannot be executed concurrently. However, we can still parallelize a sub-problem on its own. Since the boards in the output which a sub-problem will produce are independent from each other, they can be created concurrently.

Imagine boards in the input are called: $b_1, b_2, b_3 ... b_k$. For instance, inserting the digit 7 to b_2 and inserting the digit 7 to b_1 can be executed concurrently. Similarly, inserting 3 to b_i and inserting 8 to b_i can be executed concurrently. This requires deep copies of boards so that insertions do not intersect.

To achieve such parallelization, a CUDA kernel's block is tasked to check a board. The threads in that block is made responsible for inserting different digits to that board. To be more precise, if the number of boards in the input list (I) is n, n blocks are created with 9 threads. A parallel thread checks for inserting digit = ($threadIdx.x$) to the input board: $I[blockIdx.x]$. It checks whether the sums which include the insertion cell are valid. If it is valid, this arrangement of the board is placed into the output board array.

However, the number of boards that a sub-problem will produce is not known in advance. It depends on the nature of the input. Therefore, we need a dynamic parallelization method.
For this purpose, we create a controller kernel with 1 block and 1 thread.

This kernel is responsible for creating new kernels, processing the outputs, and preparing new inputs. The reason why a controller kernel is called, instead of controlling the kernels that will solve the problem from CPU, is the fact that we want to avoid data communication between CPU and GPU as much as possible for performance reasons.

The controller kernel starts with creating a kernel with 1 block and 9 threads, giving its input as the initially empty board. It waits for this kernel to finish its execution by calling:

```
cudaDeviceSynchronize()
```

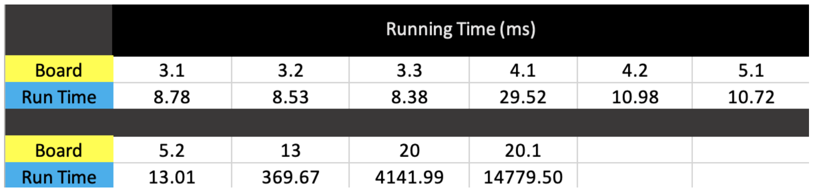
Having the child kernel's output, the controller kernel organizes the output to be the new input, deletes unnecessary memory items, allocates new memory for next kernel to produce output inside, and finally creates the new child kernel. This operation continues until a child kernel's output set is empty, or we come to the end of the board. If it comes to the end of the board successfully, it returns the final set as the set of solutions.

3 Measuring Performance

Nebula machine is used to test and measure the performance of the algorithm:

```
==prop== Running on device: 0 -- TITAN X (Pascal)
==prop== #of SM -- 28
==prop== Max Threads Per Block: -- 1024
```

Here are the running times with respect to sample boards:



	Running Time (ms)					
Board	3.1	3.2	3.3	4.1	4.2	5.1
Run Time	8.78	8.53	8.38	29.52	10.98	10.72
Board	5.2	13	20	20.1		
Run Time	13.01	369.67	4141.99	14779.50		

4 Running the Program

The following compiler command can be used to run the program:

```
nvcc kakuro_solver.cu -rdc=true
```

```
./a.out board5_2.kakuro
```