HPC Futoshiki Solver on GPU

Özgün Özerk

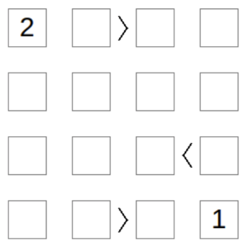
18 May 2020

1 Explanation of the Problem

In this homework, we are given the task of implementing a Futoshiki solver. The definition of the Futoshiki Puzzle and techniques for solving could be found at Futoshiki. The purpose of the assignment was not only to implement a successful solver, but also an efficient solver at the same time. The puzzles will be read from the provided input files, and the filename of the matrix must be passed to the program by command line (argc/argv). The format of the input files will be as follows:

- Each input will begin with an integer n that indicates the size of the matrix.
- Each input will contain a square matrix which cells with values presented by positive numbers and empty cells presented with **-1**'s. The values are separated with white-spaces.
- Each input will have constraints for the puzzle after the matrix. One line for one constraint. Constraints are given in a format that; the value of cell resides in coordinate of first two numbers is bigger than the cell resides in latter two. For example, constraint 1 2 1 3 means that cell at (1, 2) will have bigger value than cell at (1, 3). Therefore an input `input1.txt` will be in the format:

```
4
2 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1
1 2 1 3
3 4 3 3
4 2 4 3
```



2 Parallelism Design (necessities and optimizations)

2.1 Necessities

To achieve parallelism, first thing to do is: remove the dependencies between the recursive steps. If we can assign a Futoshiki instance to each thread, and let them work on their own copy, there will be no dependency between the threads. But when a thread finds a solution, the other threads should stop. That means, we need communication between the threads. We can do this with a "volatile __shared__" variable.

volatile let's us to tell the code "don't read this from cache, check it's original value at that moment", **__shared__** is a helping us to create a common variable that opens up the opportunity for the communication between the threads. Now, each thread should check that very shared variable to determine if they should continue or not.

Considering the capabilities of GPU (cores huge in number, but not as powerful as CPU cores), domain pruning (human-like) logic have not been applied to this solution. Instead, a brute-force method have been selected. Although I'm calling it brute-force, it's not 100% brute-force. It tries to fill the non-conflicting numbers one by one. It's not trying every number, nor generating a complete random Futoshiki instance and checking it's validity.

Recursive design for GPU is usually not preferred. The GPU code will be run by every thread, and stack is very limited per thread. If recursion is deep, the code usually does not work correctly. I have learned that the hard way :')
As a solution, unrolling of the recursion into a while loop, generally surpasses this barrier.

2.2 Optimization

Although it's not directly related to GPU Parallelization, an extra optimization has been done on the task. Most of the instances having multiple copies of the same constraints. To eliminate this, "set" data-structure have been used. Each constraint has 4 digits (as 2 pairs to represent 2 cells). Every 4 item is composed into a single 4-digit number and put into the set. Since a set cannot containt multiple copies of the same element, doubles are automatically eliminated. Then those 4-digit numbers are converted back again to their original format from the set.

```
for(int c = 0; c < constraint_sizes[i]; c++)
{
    std::getline(file, file_line);
    std::istringstream iss(file_line);
    iss >> elem0 >> elem1 >> elem2 >> elem3;
    elem4 = elem0 * 1000 + elem1 * 100 + elem2 * 10 + elem3;
    constraint_set.insert(elem4);
}
temp_size = constraint_set.size();
constraint_sizes[i] = temp_size;
constraint_beginnings[i] = sum;
sum += temp_size;
```

Coming back to the GPU: Threads from the same block can communicate very well and fast, we should design our system accordingly. A single Futoshiki instance will be given to each block. Inside that block, a copy of that instance will be created per thread. With this design, using total of 144,000 blocks, all of the Futoshiki instances can be solved in parallel.

2.2.1 Latency

The Futoshiki's we are dealing with, have 25 cells (5x5). Every thread have been assigned to a different cell, and starting the brute-force search from there. So the same instance is being tried to be solved from every single cell as a start point. Whichever thread finds the solution, modifies the shared variable, so other threads stop when they come back to the while loop as the next step of their progress.

2.2.2 Throughput

This design allows us to solve Futoshiki instances as many as our block amount in the GPU. We have currently using 60 thread per block. But without much speed penalty, this can be lowered to 25. But for efficiency, the thread number should be multitudes of 32. In this case, total thread number divided by 32 would be the maximum capacity of Futoshiki's instances that can be solved at once.

If you are curious about why 60 threads are being used by each block, the explanation is here: Different from the Sudoku, Futoshiki has constraints. These constraints have been stored in an array, and for 5x5 instances, 60 was the maximum elements of a single Futoshiki instances constraints.

Blocks have been already utilized for this design, and not using shared memory would be a bad design. Thus, we are copying each constraint element to the shared memory with threads. This can be done at maximum speed if the thread amount is equal to the array size (which is 60 at max). Of course, the bottleneck of this problem is not copying the constraints from global to shared memory. That's why, it can be done with 25 threads too (a thread to each cell). If we have more capacity in our GPU than the Futoshiki's to be solved (which will be the general case), assigning 64 threads per block is a good idea (remember, thread count should be multiple of 32) In the opposite case, 25 threads are needed, so 32 threads should be suffice per block, increasing the total numbers of Futoshiki instances can be solved by 2 times.

Although the above part sounds good in theory, when running, the performance always dropped when the thread count is below 96. This behaviour is still a mystery for me. Because of the very reason, the original version will be discussed from now on (32 thread version is discarded from this report).

In conclusion, the throughput can be summarized as

$$(N_f) = \frac{N_t}{96}$$

where N_f represents # Futoshiki Instances can be solved concurrently and N_t represents number of threads in GPU

3 The Algorithm (Explanation of the functions

Algorithm 1: recursive

```
Data: matrix, constraints, not_found_flag
Result: solution, true if the solution is found
if all cells are filled then
    not_found_flag = false;
    copy the solution to the shared memory;
    return true;
end if
for num in range(1,5) do
    if is num safe to place in that cell then
        matrix[row][col] = num;
        if recursive then
            return true;
        end if
        matrix[row][col] = -1;
    end if
end for
return false
```

This was the initial design. Unfortunately, as described above, recursive design is not a good choice for GPU.

Algorithm 2: while loop

Data: matrix, constraints, not_found_flag

Result: solution, true if the solution is found

```
while not_found_flag and kaputt = 0 do
    if all cells are filled then
        not_found_flag = false;
        copy the solution to the shared memory;
    else
        dead_end = true;
        for num in range(allowed_value,5) and dead_end = false do
            if is num safe to place in that cell then
                dead_end = false;
                allowed_value = 0; matrix[row][col] = num;
                toStack.insert(row, col, num);
            end if
        end for
        if dead_end then
            row, col, num = toStack.pop();
            if toStack.empty() then
                kaputt = 1;
                // no solution
            end if
            matrix[row][col] = -1;
        end if
    end if
end while
```

4 Performance

System and Technology specifications:

CPU:
Intel(R) Xeon(R) CPU E7-4870 v2 @ 2.30GHz **X60**

GPU:
Tesla K40c 12GB GDDR5

RAM:
516767 MB

Operating System:
CentOS 6.5

GCC Version:
10.0

GCC Version:
4.4.7

OpenMP Version:
4.5

GPU Performance on 144000 Futoshiki's

input:

144000 different Futoshiki instances have been supplied to the program via a .txt file. Each instance having the size 5x5.

Table 1: 25 Threads Assigned To Different Cells

# Threads	*Kernel Duration*	*CPU->GPU*	*GPU->CPU*
60	1631 - 1639 ms	14 ms	8 ms
64	1631 - 1639 ms	14 ms	8 ms
95	1581 - 1587 ms	14 ms	8 ms
95	1581 - 1587 ms	14 ms	14 ms
125	1581 - 1587 ms	14 ms	8 ms
320	1634- 1636 ms	14 ms	14 ms

Discussion:

Because of the design, we cannot provide less than 60 threads to the block, since 60 threads will be doing the job of copying the global constraint array into a shared one, and also 25 threads will be doing the brute-force search from different locations. The interesting thing is, the version that all of the 25 threads starting the brute-force search from the same location is a bit faster than this. The explanation might be, the extra control mechanism's are creating an overhead and the instance size of 5x5 being too small to compensate this overhead. So I've reduced the thread count to 5 for solving the Futoshiki, assigning those 5 threads to every cell of the first row. The difference is not drastically better, but it's better. Here is the table below:

Table 2: 25 Threads Assigned To Same Cell

# Threads	*Kernel Duration*	*CPU->GPU*	*GPU->CPU*
5	1367 - 1370 ms	14 ms	8 ms
25	1374 - 1376 ms	14 ms	8 ms
96	1376 - 1378 ms	14 ms	8 ms

Comparing with CPU:

Below is the table for the sequential CPU performance. Keep in mind that, CPU cores are very powerful, and for solving 5x5 matrix, latency improvement is not necessary. Furthermore, it will be probably harmful because of the parallelization overhead. However, the core count might be used for throughput. Let's be generous and take the best algorithm's average result of 5x5 inputs. When we multiply it with the Futoshiki instance amount of 144000, we get 34.56 seconds. Dividing it with 1.37, we need to have 25.2262773723 cores, and again, that's calculated with the 0 overhead of parallelism and based on the best scenario.

Input	*Size*	*Time (in seconds)*	
		OOP with roll-back	**Functional without roll-back**
input1	4x4	0.000113893	0.000107203
input1_2	4x4	0.000098604	0.000098969
input2	5x5	0.000202812	0.000223842
input2_2	5x5	0.000327289	0.00025174
input3	6x6	0.000753481	0.000703964
input3_2	6x6	0.00578789	0.00506186
input4	7x7	0.00875532	0.00870612
input4_2	7x7	0.0310422	0.0260722
input5	8x8	1.64324	1.43945
input5_2	8x8	3.23352	2.80053

Despite all the calculations above, this comparison is not accurate for a general comparison of the GPU and CPU performance, since we haven't utilized the GPU's all threads. Theoretically, almost in the same time for 144,000 instances, GPU could solve much more instances of Futoshiki. Thus, achieving the same performance with the CPU would require significantly more cores. For that, we might need billions of different Futoshiki instances. But since that conditions haven't met, the aforesaid comparison between GPU and CPU will not be discussed in this report.

5 How to Run and Compile

The input file to be read and the cu file should be in the same folder. For compiling, open a terminal in the respective folder, type the command "module load cuda/10.0" for loading the necessary setup, then "nvcc filename.cu" for compiling, and lastly "./a.out input.txt" for running it. The timing results should be displayed in the terminal, and solutions of the Futoshiki's can be found under solution.txt file in the same folder.