Kakuro Solver on GPU with CUDA

Homework 3

Rebah Özkoç 29207

CS 406 Parallel Computing

May 28, 2023

Introduction

In this homework, I tried to use CUDA to improve the performance of the serial Kakuro Solver. First, I started with the previous recursive backtracking algorithm. Then I converted it to an iterative backtracking algorithm with stacks. After that, I tried to use GPU to speed up some parts of the algorithm.

Problems with Parallelizing Backtracking Algorithm:

The recursive backtracking algorithm to solve Kakuro simply works like this: The algorithm starts to try a candidate value for a cell. If the value is suitable for that cell at the moment, the algorithm partially assigns it to that cell and continues to execute with the next empty cell. If a cell cannot be filled with any value, the algorithm goes back one step and tries to assign a new value to that cell. The execution continues until a solution is found or it is shown that no assignments can be done to the board. If we examine the algorithm, we can see that the search for the solution is a depth-first search algorithm, and the depth-first search is hard to parallelize because the search depends on the results of upper-level searches.

Why does Using Stacks not Work?

The GPU kernels are not returning any value and each thread is running the same code. The recursive algorithms are not suitable for GPU because the result of a function depends on another function's result. At first glance, it looks like using a stack and implementing a to iterative backtracking algorithm with stacks solves the problem. However, this approach does not remove the negative sides of the recursion. In a stack, we can only access the top element, and this means we can only process one element at a time. This gives practically the same performance as using a serial implementation.

On the other hand, if try to access deeper elements we break the logic of the backtracking algorithm. Another issue with the stacks is they need to have variable lengths and allocating variables with variable lengths on GPU increases the memory bandwidth usage and the performance of the application drops.

Bread-First Search Solution:

We should modify our function to get rid of the depth-first search and start to use a breadth-first search algorithm. In this case, we can check all the possible values for a cell in a given time and we can benefit from parallelizing.

For example, we can start from the first empty cell, and we can try all the numbers from 1 to 9 in a different thread. After that, we can continue with each of the valid assignments for that cell and try the values for the next empty cell.

In this algorithm, we need to make copies of boards in each step. As the worst case at the end of the first step, we will have 9 different boards and in the second step, this increases

to 9*8 = 72 boards. For a board with size 4x4, we will have 16 cells and in the worst case, we will have 9^{16} boards roughly.

We can solve the memory problem by freeing the invalid board spaces at each execution.

Pseudo Code of the Algorithm:

1. Implement a breadth-first search from the initial Kakuro board to determine all potential boards with the first empty space occupied with a partial assignment. For each value create a new board copy.

2. Attempt to solve each of the new boards separately in different threads on the GPU. If a solution is found, terminate the program and return the solution.

However, this solution was too complicated, and I was not able to fully implement it in CUDA. Instead, I tried to speed up my code by performing the sum checks for all sums in GPU.

Performance:

During the testing of my program, Nebula was not working properly and I tested the code on my local environment.

I used CUDA 12.1.1 and Visual Studio 2019 as software.

My computer has Nvidia GeForce GTX 1050 and Intel Core I7-8750H.

The algorithm that I implemented is working correctly but initializing the GPU kernels and copying the matrix at each step takes some time, thus the speed of the code for smaller boards is worse than the serial execution. For the board 20_1 and bigger, the memory of my GPU is not sufficient, and the program is not able to solve the problem.

Execution times:

Board 3_1: 0.013661 seconds

Board 3_2: 0.034123 seconds

Board 4_1: 0.078314 seconds

Board 4_2: 0.133789 seconds

Board 5_1: 0.185275 seconds

Board 5_2: 0.222925 seconds

Board 20_1: Not available

...

How to Compile:

```
nvcc kakuro_solver_hw3.cu -Xcompiler -fopenmp -Xcompiler -O3 -O3
./a.out board_name.kakuro
```