CS 531 - HW3

Ege Alpay

May 2020

1 The Code

In this homework, we were asked to implement Futhoshiki solver which will run on GPU. Running code on GPU is not straightforward as running it on CPU. Data should be transfered between CPU and GPU, before and after the execution.

```
    int *d_grids;
    int *d_constraints;
    int *d_constraint_sizes;

    cudaMalloc((void **) &d_grids, sizeof(int) * no_grids *
    ↪  SIZE * SIZE);
    cudaMalloc((void **) &d_constraints, sizeof(int) *
    ↪  NUM_CONSTRAINTS_ELEMENTS);
    cudaMalloc((void **) &d_constraint_sizes, sizeof(int) *
    ↪  no_grids);

    cudaMemcpy(d_grids, grids_1d, sizeof(int) * no_grids * SIZE
    ↪  * SIZE, cudaMemcpyHostToDevice);
    cudaMemcpy(d_constraints, constraints_1d, sizeof(int) *
    ↪  NUM_CONSTRAINTS_ELEMENTS, cudaMemcpyHostToDevice);
    cudaMemcpy(d_constraint_sizes, constraint_sizes,
    ↪  sizeof(int) * no_grids, cudaMemcpyHostToDevice);

    int threadPerBlock = 64;
    int blockCount = (no_grids / threadPerBlock) + 1;
```

```
    solve <<< blockCount, threadPerBlock >>> (d_grids,
    ↪ d_constraints, d_constraint_sizes, no_grids);
    cudaDeviceSynchronize();

    cudaMemcpy(grids_1d, d_grids, sizeof(int) * no_grids * SIZE
    ↪ * SIZE, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    cudaFree(d_grids);
    cudaFree(d_constraints);
    cudaFree(d_constraint_sizes);
```

In the code above, you can find the way of running code on GPU. In the beginner code, we had *grids* variable as 3D array and *constraints* variable as 2D array. I have read that transferring these types of array are not recommended, so I have flatten these arrays into 1D. First, memory allocation is done, then data is transferred from CPU to CPU. Later on, kernel is called, you can find the implementation below, then data is transferred back to CPU from GPU. Lastly, memory is deallocated on GPU.

```
__global__ void solve(int *grids, int *constraints, int
↪ *constraint_sizes, int gridCount) {
    int futoshiki[25];
    int constraintSizeForPuzzle;

    int globalId = blockIdx.x * blockDim.x + threadIdx.x;

    if (globalId < gridCount) {
        constraintSizeForPuzzle = constraint_sizes[globalId];

        int localConstraints[60];

        for (int i = 0; i < 25; i++) {
            futoshiki[i] = grids[globalId * 25 + i];
        }

        int constraintStartIndex = 0;
        for (int i = 0; i < globalId; i++) {
```

```
18              constraintStartIndex += constraint_sizes[i] * 4;
19          }
20
21          for (int i = 0; i < constraintSizeForPuzzle; i++) {
22              localConstraints[4 * i] =
                ↪  constraints[constraintStartIndex + 4 * i];
23              localConstraints[4 * i + 1] =
                ↪  constraints[constraintStartIndex + 4 * i + 1];
24              localConstraints[4 * i + 2] =
                ↪  constraints[constraintStartIndex + 4 * i + 2];
25              localConstraints[4 * i + 3] =
                ↪  constraints[constraintStartIndex + 4 * i + 3];
26          }
27
28          solveSingleThread(futoshiki, localConstraints,
            ↪  constraint_sizes[globalId]);
29
30          for (int i = 0; i < 25; i++) {
31              grids[globalId * 25 + i] = futoshiki[i];
32          }
33      }
34  }
```

Function *solve* is the first function triggered on GPU. Global Id of a thread is calculated first. Then, each thread will get it's puzzle and corresponding constraints with respect to their global id. Later on, each thread will call *solveSingleThread* function by passing it's puzzle and constraints. This function is basically solves the given puzzle with backtracking. However, it does not use recursion since I have read that it is not so efficient, and memory problems occurs because of the depth. Lastly, the solved puzzle will be written back to *grids* variable. All these indexing operations are calculated based on global id of the thread.
To sum up, each thread will be working on a unique puzzle where number of the threads is equal to number of puzzles.

2 Execution times

Duration for Memory Allocation & Deallocation in GPU and data movement between GPU & CPU is not dependent to block size or thread count. You can find the duration in seconds, in the following table:

Operation	Execution Time in Seconds
Memory Allocations for GPU	0.0007
CPU to GPU Data Transfer	0.014
GPU to CPU Data Transfer	0.008
Memory Deallocation for GPU	0.004

In Cuda Programming, a block can have maximum 1024 threads. For that reason, I have tried different number of threads to observe execution time.

Number of Threads Per Block	Kernel Execution Time in Seconds
1	4.972
2	3.569
4	2.820
8	2.473
16	2.290
32	2.200
64	1.847
128	1.853
256	1.867
512	1.889
1024	1.906

Fastest execution time was obtained where 64 Threads were used in a block where number of blocks is equal to $no\_grids/64$.

3 Tricks Done for Efficiency

Transferring non 1D array needs multiple *cudaMemCpy* calls. In order to reduce number of calls of *cudaMemCpy*, these types of arrays are flatten into 1D arrays. So, we were able to transfer data with only one *cudaMemCpy* call.

4 Compile and Run

For development, I have used CLion IDE to write the code. For compilation and testing the implementation, I have used gcc 9.2.0 on my Linux Mint machine. On Gandalf, gcc 4.4.7 with cuda 10.0 was used.
Following command was used to run the code on Gandalf:

```
1    module load cuda/10.0.
2    nvcc solution.cu
3    ./a.out input.txt
```