Computational Science on Many-Core Architectures: Exercise 10

Viktor Beck, 11713110

january 2023

Divoc simulator

a)

```
// for a)
float random_function(){
  return (float) rand()/RAND_MAX;
}
```

```
  // ########## TASK a) ##########
  // create random points and copy them over to GPU
  double *random_vec = (double *)malloc(sizeof(double) * N);
  std::generate(random_vec, random_vec + N, random_function);

  double *cuda_random_vec;
  cudaMalloc(&cuda_random_vec, sizeof(double) * N);
  cudaMemcpy(cuda_random_vec, random_vec, sizeof(double) * N,
      cudaMemcpyHostToDevice);
```

The sequence should be long enough such that the numbers are not repeating.

b)

For the generation of pseudo random numbers the linear congruential generator was implemented. It general, this is a rather poor RNG but it should be enough to test this simulation. It works as follows: a starting value is set with the thread id and a seed we get as input. Each thread will then calculate one or more numbers of the random vector which will be the result. Since every thread has a different value for `s` the modulo function will produce "random" values. In the end, the values are normalized to numbers between 0 and 1. The same seed will of course produce the same numbers.

```
// ########## TASK b) ##########

// linear congruential generator (poor rng but should be enough)
__global__ void LCG(double *random_vec, int N, int seed)
{
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // individual starting value for each thread
  int s = seed + id;
  double tmp;
  for (int i = id; i < N; i += blockDim.x * gridDim.x)
```

```
  {
    s = s * 1234567891 + 54321; // some random parameters
    tmp = (s % 100);
    //printf("%g\n", tmp);
    random_vec[i] = tmp / 200 + 0.5f; // s.t. we get numbers between 0 and 1
    printf("%g\n", random_vec[i]);
  }
}
```

Regarding the performance, this function will likely be faster than the one from a), especially, for larger vector sizes because of parallelization but more importantly because copying from CPU to GPU presents a bottleneck (for larger vectors).

c)

The initialization was ported to the GPU by calling the `init_input` and `init_output` functions (with `__device__`) in a CUDA kernel that was executed by one thread. It could have also been initialized in the same kernel as the simulation by every thread calling the functions. To make it work it is necessary to copy the structs over to the GPU.

```
// ########## TASK c) ##########

__global__ void initialize(SimInput_t *input, SimOutput_t *output)
{
  init_input(input);
  printf("TEST: mask_threshold %d\n", input->mask_threshold);
  init_output(output, (int) &input->population_size);
}
```

Unfortunately, I was not able to get the simulation phase running on the GPU. The plan was to rewrite the `rand()` calls in the `run_simulation` function such that a thread gets a random number from the random vector generated in task b). Each time the random vector is accessed, a counter would be increased such that the same number is not taken twice. Since the ordering of the threads can be random at some times, real (or at least "realer") randomness could have been introduced into the function. I am not sure if this would be a promising approach but I was not able to test it accordingly since the simulation part gave me a segmentation fault all the time (I couldn't figure out for two days what's wrong so I gave up). I first wanted to make it run on the CPU with the random numbers taken from the random vector and then make it run in the GPU. You can see my approach in the `run` function.

Also, I had some weird things happening when using structs in kernels which made finding mistakes even harder. For example, some things would not be printed even when the compilation was successful and when I deleted something completely not related to the thing I wanted to print then it was printed again??? Some other students had similar problems with that.

Appendix

```
/**
 * 360.252 - Computational Science on Many-Core Architectures
 * WS 2022/23, TU Wien
 *
 * Simplistic simulator for a disease of very immediate concern (DIVOC). Inspired by COVID-19 simulations.
 *
```

```
 * DISCLAIMER: This simulator is for educational purposes only.
 * It may be arbitrarily inaccurate and should not be used for drawing any
     conclusions about any actual virus.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "timer.hpp"
#include <vector>
#include <algorithm>
#include <iostream>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define YEAR 10

//
// Data container for simulation input
//

typedef struct
{

  int population_size; // Number of people to simulate

  //// Configuration
  int mask_threshold;      // Number of cases required for masks
  int lockdown_threshold;  // Number of cases required for lockdown
  int infection_delay;     // Number of days before an infected person can pass
      on the disease
  int infection_days;      // Number of days an infected person can pass on the
      disease
  int starting_infections; // Number of infected people at the start of the year
  int immunity_duration;   // Number of days a recovered person is immune

  // for each day:
  int *contacts_per_day;            // number of other persons met each day to
      whom the disease may be passed on
  double *transmission_probability; // how likely it is to pass on the infection
      to another person

} SimInput_t;

__host__ __device__ void init_input(SimInput_t *input)
{
  input->population_size = 8916845; // Austria's population in 2020 according to
      Statistik Austria

  input->mask_threshold = 5000;
  input->lockdown_threshold = 50000;
  input->infection_delay = 5; // 5 to 6 days incubation period (average)
      according to WHO
  input->infection_days = 3;  // assume three days of passing on the disease
  input->starting_infections = 10;
  input->immunity_duration = 180; // half a year of immunity

  input->contacts_per_day = (int *)malloc(sizeof(int) * YEAR);
  input->transmission_probability = (double *)malloc(sizeof(double) * YEAR);
  for (int day = 0; day < YEAR; ++day)
```

```c
  {
    input->contacts_per_day[day] = 6;                                                    // arbitrary assumption of six possible transmission contacts per person per day, all year
    input->transmission_probability[day] = 0.2 + 0.1 * cos((day / (double) YEAR) * 2 * M_PI); // higher transmission in winter, lower transmission during summer
  }
}

typedef struct
{
  // for each day:
  int *active_infections; // number of active infected on that day (including incubation period)
  int *lockdown;          // 0 if no lockdown on that day, 1 if lockdown

  // for each person:
  int *is_infected; // 0 if healty, 1 if currently infected
  int *infected_on; // day of infection. negative if not yet infected. January 1 is Day 0.

} SimOutput_t;

//
// Initializes the output data structure (values to zero, allocate arrays)
//
__host__ __device__ void init_output(SimOutput_t *output, int population_size)
{
  output->active_infections = (int *)malloc(sizeof(int) * YEAR);
  output->lockdown = (int *)malloc(sizeof(int) * YEAR);
  for (int day = 0; day < 10; ++day)
  {
    output->active_infections[day] = 0;
    output->lockdown[day] = 0;
  }

  output->is_infected = (int *)malloc(sizeof(int) * population_size);
  output->infected_on = (int *)malloc(sizeof(int) * population_size);

  for (int i = 0; i < population_size; ++i)
  {
    output->is_infected[i] = 0;
    output->infected_on[i] = 0;
  }
}

void run_simulation(const SimInput_t *input, SimOutput_t *output)
{
  //
  // Init data. For simplicity we set the first few people to 'infected'
  //
  for (int i = 0; i < input->population_size; ++i)
  {
    output->is_infected[i] = (i < input->starting_infections) ? 1 : 0;
    output->infected_on[i] = (i < input->starting_infections) ? 0 : -1;
  }

  //
  // Run simulation
  //
  for (int day = 0; day < YEAR; ++day) // loop over all days of the year
```

```c
  {
    //
    // Step 1: determine number of infections and recoveries
    //
    int num_infected_current = 0;
    int num_recovered_current = 0;
    for (int i = 0; i < input->population_size; ++i)
    {

      if (output->is_infected[i] > 0)
      {
        if (output->infected_on[i] > day - input->infection_delay - input->infection_days &&
            output->infected_on[i] <= day - input->infection_delay) // currently infected and incubation period over
          num_infected_current += 1;
        else if (output->infected_on[i] < day - input->infection_delay - input->infection_days)
          num_recovered_current += 1;
      }
    }

    output->active_infections[day] = num_infected_current;
    if (num_infected_current > input->lockdown_threshold)
    {
      output->lockdown[day] = 1;
    }
    if (day > 0 && output->lockdown[day - 1] == 1)
    { // end lockdown if number of infections has reduced significantly
      output->lockdown[day] = (num_infected_current < input->lockdown_threshold / 3) ? 0 : 1;
    }
    char lockdown[] = " [LOCKDOWN]";
    char normal[] = "";
    printf("Day %d%s: %d active, %d recovered\n", day, output->lockdown[day] ? lockdown : normal, num_infected_current, num_recovered_current);

    //
    // Step 2: determine today's transmission probability and contacts based on pandemic situation
    //
    double contacts_today = input->contacts_per_day[day];
    double transmission_probability_today = input->transmission_probability[day];
    if (num_infected_current > input->mask_threshold)
    { // transmission is reduced with masks. Arbitrary factor: 2
      transmission_probability_today /= 2.0;
    }
    if (output->lockdown[day])
    { // contacts are significantly reduced in lockdown. Arbitrary factor: 4
      contacts_today /= 4;
    }

    //
    // Step 3: pass on infections within population
    //
    for (int i = 0; i < input->population_size; ++i) // loop over population
    {
      if (output->is_infected[i] > 0 && output->infected_on[i] > day - input->infection_delay - input->infection_days // currently infected
          && output->infected_on[i] <= day - input->infection_delay)                   // already infectious
```

```
      {
        // pass on infection to other persons with transmission probability
        for (int j = 0; j < contacts_today; ++j)
        {
          double r = ((double)rand()) / (double)RAND_MAX; // random number between 0 and 1
          if (r < transmission_probability_today)
          {
            r = ((double)rand()) / (double)RAND_MAX; // new random number to determine a random other person to transmit the virus to
            int other_person = r * input->population_size;
            if (output->is_infected[other_person] == 0                                   // other person is not infected
                || output->infected_on[other_person] < day - input->immunity_duration) // other person has no more immunity
            {
              output->is_infected[other_person] = 1;
              output->infected_on[other_person] = day;
            }
          }

        } // for contacts_per_day
      }   // if currently infected
    }     // for i

  } // for day
}


// ########## TASK b) ##########

// linear congruential generator (poor rng but should be enough)
__global__ void LCG(double *random_vec, int N, int seed)
{
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // individual starting value for each thread
  int s = seed + id;
  double tmp;
  for (int i = id; i < N; i += blockDim.x * gridDim.x)
  {
    s = s * 1234567891 + 54321; // some random parameters
    tmp = (s % 100);
    //printf("%g\n", tmp);
    random_vec[i] = tmp / 200 + 0.5f; // s.t. we get numbers between 0 and 1
    printf("%g\n", random_vec[i]);
  }
}

// ########## TASK c) ##########

__global__ void initialize(SimInput_t *input, SimOutput_t *output)
{
  init_input(input);
  printf("TEST: mask_threshold %d\n", input->mask_threshold);
  init_output(output, (int) &input->population_size);
}

// test if input has been initialized correctly
__global__ void test_init(SimInput_t *input, SimOutput_t *output)
{
  printf("TEST: mask_threshold %d\n", input->mask_threshold);
```

```
}

//__global__
void run(SimInput_t *input, SimOutput_t *output, double *random_vec)
{
  printf("\nGPU RUN STARTS HERE\n");

  int rng_count = 0;
  //
  // Init data. For simplicity we set the first few people to 'infected'
  //
  for (int i = 0; i < input->population_size; ++i)
  {
    output->is_infected[i] = (i < input->starting_infections) ? 1 : 0;
    output->infected_on[i] = (i < input->starting_infections) ? 0 : -1;
  }

  //
  // Run simulation
  //
  for (int day = 0; day < YEAR; ++day) // loop over all days of the year
  {
    //
    // Step 1: determine number of infections and recoveries
    //
    int num_infected_current = 0;
    int num_recovered_current = 0;
    for (int i = 0; i < input->population_size; ++i)
    {

      if (output->is_infected[i] > 0)
      {
        if (output->infected_on[i] > day - input->infection_delay - input->
            infection_days &&
            output->infected_on[i] <= day - input->infection_delay) // currently
                infected and incubation period over
          num_infected_current += 1;
        else if (output->infected_on[i] < day - input->infection_delay - input->
            infection_days)
          num_recovered_current += 1;
      }
    }

    output->active_infections[day] = num_infected_current;
    if (num_infected_current > input->lockdown_threshold)
    {
      output->lockdown[day] = 1;
    }
    if (day > 0 && output->lockdown[day - 1] == 1)
    { // end lockdown if number of infections has reduced significantly
      output->lockdown[day] = (num_infected_current < input->lockdown_threshold /
           3) ? 0 : 1;
    }
    char lockdown[] = " [LOCKDOWN]";
    char normal[] = "";
    printf("Day %d%s: %d active, %d recovered\n", day, output->lockdown[day] ?
        lockdown : normal, num_infected_current, num_recovered_current);
    printf("Random numbers so far: %d, Test rng number: %g\n", rng_count,
        random_vec[rng_count]);
    //
    // Step 2: determine today's transmission probability and contacts based on
        pandemic situation
```

```
    //
    double contacts_today = input->contacts_per_day[day];
    double transmission_probability_today = input->transmission_probability[day];
    if (num_infected_current > input->mask_threshold)
    { // transmission is reduced with masks. Arbitrary factor: 2
      transmission_probability_today /= 2.0;
    }
    if (output->lockdown[day])
    { // contacts are significantly reduced in lockdown. Arbitrary factor: 4
      contacts_today /= 4;
    }

    //
    // Step 3: pass on infections within population
    //
    for (int i = 0; i < input->population_size; ++i) // loop over population
    {
      if (output->is_infected[i] > 0 && output->infected_on[i] > day - input->infection_delay - input->infection_days // currently infected
          && output->infected_on[i] <= day - input->infection_delay) // already infectious
      {
        // pass on infection to other persons with transmission probability
        for (int j = 0; j < contacts_today; ++j)
        {
          double r = random_vec[rng_count++]; // random number between 0 and 1
          if (r < transmission_probability_today)
          {
            r = random_vec[rng_count++]; // new random number to determine a random other person to transmit the virus to
            int other_person = r * input->population_size;
            if (output->is_infected[other_person] == 0 // other person is not infected
                || output->infected_on[other_person] < day - input->immunity_duration) // other person has no more immunity
            {
              output->is_infected[other_person] = 1;
              output->infected_on[other_person] = day;
            }
          }

        } // for contacts_per_day
      }   // if currently infected
    }     // for i

  } // for day
}

// for a)
float random_function(){
  return (float) rand()/RAND_MAX;
}

int main(int argc, char **argv)
{
  int N_BLOCKS = 256;
  int N_THREADS = 256;

  // temporary
  int N = 10;

```

```
// ########## TASK a) ##########
// create random points and copy them over to GPU
double *random_vec = (double *)malloc(sizeof(double) * N);
std::generate(random_vec, random_vec + N, random_function);

double *cuda_random_vec;
cudaMalloc(&cuda_random_vec, sizeof(double) * N);
cudaMemcpy(cuda_random_vec, random_vec, sizeof(double) * N,
    cudaMemcpyHostToDevice);

// ########## TASK b) ##########
// random_vec will be overwritten by this function
LCG<<<N_BLOCKS, N_THREADS>>>(cuda_random_vec, N, 42);

// ########## TASK c) ##########
// create input and output strucs on CPU and copy them over to GPU
SimInput_t input;
SimOutput_t output;
SimInput_t *cuda_input;
cudaMalloc(&cuda_input, sizeof(SimInput_t));
SimOutput_t *cuda_output;
cudaMalloc(&cuda_output, sizeof(SimOutput_t));

cudaMemcpy(cuda_input, &input, sizeof(SimInput_t), cudaMemcpyHostToDevice);
cudaMemcpy(cuda_output, &output, sizeof(SimOutput_t), cudaMemcpyHostToDevice);

// let a single thread do the initialization
initialize<<<1, 1>>>(cuda_input, cuda_output);

cudaDeviceSynchronize();


// test if initialize function worked
test_init<<<1,1>>>(cuda_input, cuda_output);

cudaDeviceSynchronize();

// cudaMemcpy(&input, cuda_input, sizeof(SimInput_t), cudaMemcpyDeviceToHost);


init_input(&input);
init_output(&output, input.population_size);

Timer timer;
srand(0); // initialize random seed for deterministic output
timer.reset();
run_simulation(&input, &output);
printf("Simulation time: %g\n", timer.get());

cudaDeviceSynchronize();

//init_input(&input);
//init_output(&output, input.population_size);

// tried to make it work at first on CPU but segmentation fault ...
//run(&input, &output, random_vec);

// next step would have been to make it run with one thread on GPU
//run<<<1,1>>>(cuda_input, cuda_output, cuda_random_vec);
```

```
  return EXIT_SUCCESS;
}
```