

Computational Science on Many-Core Architectures:

Exercise 10

Viktor Beck, 11713110

january 2023

Divoc simulator

a)

```
1 // for a)
2 float random_function(){
3     return (float) rand()/RAND_MAX;
4 }

1 // ##### TASK a) #####
2 // create random points and copy them over to GPU
3 double *random_vec = (double *)malloc(sizeof(double) * N);
4 std::generate(random_vec, random_vec + N, random_function);
5
6 double *cuda_random_vec;
7 cudaMalloc(&cuda_random_vec, sizeof(double) * N);
8 cudaMemcpy(cuda_random_vec, random_vec, sizeof(double) * N,
9             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. In 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.

```
1 // ##### TASK b) #####
2
3 // linear congruential generator (poor rng but should be enough)
4 __global__ void LCG(double *random_vec, int N, int seed)
5 {
6     int id = blockIdx.x * blockDim.x + threadIdx.x;
7
8     // individual starting value for each thread
9     int s = seed + id;
10    double tmp;
11    for (int i = id; i < N; i += blockDim.x * gridDim.x)
```

```

12 {
13     s = s * 1234567891 + 54321; // some random parameters
14     tmp = (s % 100);
15     //printf("%g\n", tmp);
16     random_vec[i] = tmp / 200 + 0.5f; // s.t. we get numbers between 0 and 1
17     printf("%g\n", random_vec[i]);
18 }
19 }

```

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.

```

1 // ##### TASK c) #####
2
3 __global__ void initialize(SimInput_t *input, SimOutput_t *output)
4 {
5     init_input(input);
6     printf("TEST: mask_threshold %d\n", input->mask_threshold);
7     init_output(output, (int) &input->population_size);
8 }

```

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

```

1 /**
2  * 360.252 - Computational Science on Many-Core Architectures
3  * WS 2022/23, TU Wien
4  *
5  * Simplistic simulator for a disease of very immediate concern (DIVOC). Inspired
6  *   by COVID-19 simulations.
7  */

```

```

7  * DISCLAIMER: This simulator is for educational purposes only.
8  * It may be arbitrarily inaccurate and should not be used for drawing any
   conclusions about any actual virus.
9  */
10
11 #include <stdlib.h>
12 #include <stdio.h>
13 #include <math.h>
14 #include "timer.hpp"
15 #include <vector>
16 #include <algorithm>
17 #include <iostream>
18
19 #ifndef M_PI
20 #define M_PI 3.14159265358979323846
21 #endif
22
23 #define YEAR 10
24
25 //
26 // Data container for simulation input
27 //
28
29 typedef struct
30 {
31
32     int population_size; // Number of people to simulate
33
34     /// Configuration
35     int mask_threshold; // Number of cases required for masks
36     int lockdown_threshold; // Number of cases required for lockdown
37     int infection_delay; // Number of days before an infected person can pass
   on the disease
38     int infection_days; // Number of days an infected person can pass on the
   disease
39     int starting_infections; // Number of infected people at the start of the year
40     int immunity_duration; // Number of days a recovered person is immune
41
42     // for each day:
43     int *contacts_per_day; // number of other persons met each day to
   whom the disease may be passed on
44     double *transmission_probability; // how likely it is to pass on the infection
   to another person
45
46 } SimInput_t;
47
48 __host__ __device__ void init_input(SimInput_t *input)
49 {
50     input->population_size = 8916845; // Austria's population in 2020 according to
   Statistik Austria
51
52     input->mask_threshold = 5000;
53     input->lockdown_threshold = 50000;
54     input->infection_delay = 5; // 5 to 6 days incubation period (average)
   according to WHO
55     input->infection_days = 3; // assume three days of passing on the disease
56     input->starting_infections = 10;
57     input->immunity_duration = 180; // half a year of immunity
58
59     input->contacts_per_day = (int *)malloc(sizeof(int) * YEAR);
60     input->transmission_probability = (double *)malloc(sizeof(double) * YEAR);
61     for (int day = 0; day < YEAR; ++day)

```

```

62     {
63         input->contacts_per_day[day] = 6;
64         // 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
65     }
66 }
67
68 typedef struct
69 {
70     // for each day:
71     int *active_infections; // number of active infected on that day (including
        // incubation period)
72     int *lockdown; // 0 if no lockdown on that day, 1 if lockdown
73
74     // for each person:
75     int *is_infected; // 0 if healthy, 1 if currently infected
76     int *infected_on; // day of infection. negative if not yet infected. January 1
        // is Day 0.
77 } SimOutput_t;
78
79 //
80 // Initializes the output data structure (values to zero, allocate arrays)
81 //
82 //
83 __host__ __device__ void init_output(SimOutput_t *output, int population_size)
84 {
85     output->active_infections = (int *)malloc(sizeof(int) * YEAR);
86     output->lockdown = (int *)malloc(sizeof(int) * YEAR);
87     for (int day = 0; day < 10; ++day)
88     {
89         output->active_infections[day] = 0;
90         output->lockdown[day] = 0;
91     }
92
93     output->is_infected = (int *)malloc(sizeof(int) * population_size);
94     output->infected_on = (int *)malloc(sizeof(int) * population_size);
95
96     for (int i = 0; i < population_size; ++i)
97     {
98         output->is_infected[i] = 0;
99         output->infected_on[i] = 0;
100     }
101 }
102
103 void run_simulation(const SimInput_t *input, SimOutput_t *output)
104 {
105     //
106     // Init data. For simplicity we set the first few people to 'infected'
107     //
108     for (int i = 0; i < input->population_size; ++i)
109     {
110         output->is_infected[i] = (i < input->starting_infections) ? 1 : 0;
111         output->infected_on[i] = (i < input->starting_infections) ? 0 : -1;
112     }
113
114     //
115     // Run simulation
116     //
117     for (int day = 0; day < YEAR; ++day) // loop over all days of the year

```

```

118 {
119 //
120 // Step 1: determine number of infections and recoveries
121 //
122 int num_infected_current = 0;
123 int num_recovered_current = 0;
124 for (int i = 0; i < input->population_size; ++i)
125 {
126
127     if (output->is_infected[i] > 0)
128     {
129         if (output->infected_on[i] > day - input->infection_delay - input->
            infection_days &&
130             output->infected_on[i] <= day - input->infection_delay) // currently
                infected and incubation period over
131             num_infected_current += 1;
132         else if (output->infected_on[i] < day - input->infection_delay - input->
            infection_days)
133             num_recovered_current += 1;
134     }
135 }
136
137 output->active_infections[day] = num_infected_current;
138 if (num_infected_current > input->lockdown_threshold)
139 {
140     output->lockdown[day] = 1;
141 }
142 if (day > 0 && output->lockdown[day - 1] == 1)
143 { // end lockdown if number of infections has reduced significantly
144     output->lockdown[day] = (num_infected_current < input->lockdown_threshold /
        3) ? 0 : 1;
145 }
146 char lockdown[] = " [LOCKDOWN]";
147 char normal[] = "";
148 printf("Day %d%s: %d active, %d recovered\n", day, output->lockdown[day] ?
        lockdown : normal, num_infected_current, num_recovered_current);
149
150 //
151 // Step 2: determine today's transmission probability and contacts based on
        pandemic situation
152 //
153 double contacts_today = input->contacts_per_day[day];
154 double transmission_probability_today = input->transmission_probability[day];
155 if (num_infected_current > input->mask_threshold)
156 { // transmission is reduced with masks. Arbitrary factor: 2
157     transmission_probability_today /= 2.0;
158 }
159 if (output->lockdown[day])
160 { // contacts are significantly reduced in lockdown. Arbitrary factor: 4
161     contacts_today /= 4;
162 }
163
164 //
165 // Step 3: pass on infections within population
166 //
167 for (int i = 0; i < input->population_size; ++i) // loop over population
168 {
169     if (output->is_infected[i] > 0 && output->infected_on[i] > day - input->
        infection_delay - input->infection_days // currently infected
170         && output->infected_on[i] <= day - input->infection_delay)
            // already
            infectious

```

```

171     {
172         // pass on infection to other persons with transmission probability
173         for (int j = 0; j < contacts_today; ++j)
174         {
175             double r = ((double)rand()) / (double)RAND_MAX; // random number
                        between 0 and 1
176             if (r < transmission_probability_today)
177             {
178                 r = ((double)rand()) / (double)RAND_MAX; // new random number to
                        determine a random other person to transmit the virus to
179                 int other_person = r * input->population_size;
180                 if (output->is_infected[other_person] == 0
                        // other person is not infected
181                     || output->infected_on[other_person] < day - input->
                        immunity_duration) // other person has no more immunity
182                 {
183                     output->is_infected[other_person] = 1;
184                     output->infected_on[other_person] = day;
185                 }
186             }
187         } // for contacts_per_day
188     } // if currently infected
189 } // for i
190
191 } // for day
192 }
193
194
195
196 // ##### TASK b) #####
197
198 // linear congruential generator (poor rng but should be enough)
199 __global__ void LCG(double *random_vec, int N, int seed)
200 {
201     int id = blockIdx.x * blockDim.x + threadIdx.x;
202
203     // individual starting value for each thread
204     int s = seed + id;
205     double tmp;
206     for (int i = id; i < N; i += blockDim.x * gridDim.x)
207     {
208         s = s * 1234567891 + 54321; // some random parameters
209         tmp = (s % 100);
210         //printf("%g\n", tmp);
211         random_vec[i] = tmp / 200 + 0.5f; // s.t. we get numbers between 0 and 1
212         printf("%g\n", random_vec[i]);
213     }
214 }
215
216 // ##### TASK c) #####
217
218 __global__ void initialize(SimInput_t *input, SimOutput_t *output)
219 {
220     init_input(input);
221     printf("TEST: mask_threshold %d\n", input->mask_threshold);
222     init_output(output, (int) &input->population_size);
223 }
224
225 // test if input has been initialized correctly
226 __global__ void test_init(SimInput_t *input, SimOutput_t *output)
227 {
228     printf("TEST: mask_threshold %d\n", input->mask_threshold);

```

```

229 }
230
231 //__global__
232 void run(SimInput_t *input, SimOutput_t *output, double *random_vec)
233 {
234     printf("\nGPU RUN STARTS HERE\n");
235
236     int rng_count = 0;
237     //
238     // Init data. For simplicity we set the first few people to 'infected'
239     //
240     for (int i = 0; i < input->population_size; ++i)
241     {
242         output->is_infected[i] = (i < input->starting_infections) ? 1 : 0;
243         output->infected_on[i] = (i < input->starting_infections) ? 0 : -1;
244     }
245
246     //
247     // Run simulation
248     //
249     for (int day = 0; day < YEAR; ++day) // loop over all days of the year
250     {
251         //
252         // Step 1: determine number of infections and recoveries
253         //
254         int num_infected_current = 0;
255         int num_recovered_current = 0;
256         for (int i = 0; i < input->population_size; ++i)
257         {
258
259             if (output->is_infected[i] > 0)
260             {
261                 if (output->infected_on[i] > day - input->infection_delay - input->
infection_days &&
262                     output->infected_on[i] <= day - input->infection_delay) // currently
infected and incubation period over
263                     num_infected_current += 1;
264                 else if (output->infected_on[i] < day - input->infection_delay - input->
infection_days)
265                     num_recovered_current += 1;
266             }
267         }
268
269         output->active_infections[day] = num_infected_current;
270         if (num_infected_current > input->lockdown_threshold)
271         {
272             output->lockdown[day] = 1;
273         }
274         if (day > 0 && output->lockdown[day - 1] == 1)
275         { // end lockdown if number of infections has reduced significantly
276             output->lockdown[day] = (num_infected_current < input->lockdown_threshold /
3) ? 0 : 1;
277         }
278         char lockdown[] = " [LOCKDOWN]";
279         char normal[] = "";
280         printf("Day %d%s: %d active, %d recovered\n", day, output->lockdown[day] ?
lockdown : normal, num_infected_current, num_recovered_current);
281         printf("Random numbers so far: %d, Test rng number: %g\n", rng_count,
random_vec[rng_count]);
282         //
283         // Step 2: determine today's transmission probability and contacts based on
pandemic situation

```

```

284 //
285 double contacts_today = input->contacts_per_day[day];
286 double transmission_probability_today = input->transmission_probability[day];
287 if (num_infected_current > input->mask_threshold)
288 { // transmission is reduced with masks. Arbitrary factor: 2
289     transmission_probability_today /= 2.0;
290 }
291 if (output->lockdown[day])
292 { // contacts are significantly reduced in lockdown. Arbitrary factor: 4
293     contacts_today /= 4;
294 }
295
296 //
297 // Step 3: pass on infections within population
298 //
299 for (int i = 0; i < input->population_size; ++i) // loop over population
300 {
301     if (output->is_infected[i] > 0 && output->infected_on[i] > day - input->
302         infection_delay - input->infection_days // currently infected
303         && output->infected_on[i] <= day - input->infection_delay) // already
304             infectious
305     {
306         // pass on infection to other persons with transmission probability
307         for (int j = 0; j < contacts_today; ++j)
308         {
309             double r = random_vec[rng_count++]; // random number between 0 and 1
310             if (r < transmission_probability_today)
311             {
312                 r = random_vec[rng_count++]; // new random number to determine a
313                 random other person to transmit the virus to
314                 int other_person = r * input->population_size;
315                 if (output->is_infected[other_person] == 0
316                     // other person is not infected
317                     || output->infected_on[other_person] < day - input->
318                     immunity_duration) // other person has no more immunity
319                 {
320                     output->is_infected[other_person] = 1;
321                     output->infected_on[other_person] = day;
322                 }
323             }
324         } // for contacts_per_day
325     } // if currently infected
326 } // for i
327
328 } // for day
329 }
330
331 // for a)
332 float random_function(){
333     return (float) rand()/RAND_MAX;
334 }
335
336 int main(int argc, char **argv)
337 {
338     int N_BLOCKS = 256;
339     int N_THREADS = 256;
340
341     // temporary
342     int N = 10;

```



```

340 // ##### TASK a) #####
341 // create random points and copy them over to GPU
342 double *random_vec = (double *)malloc(sizeof(double) * N);
343 std::generate(random_vec, random_vec + N, random_function);
344
345 double *cuda_random_vec;
346 cudaMalloc(&cuda_random_vec, sizeof(double) * N);
347 cudaMemcpy(cuda_random_vec, random_vec, sizeof(double) * N,
348             cudaMemcpyHostToDevice);
349
350 // ##### TASK b) #####
351 // random_vec will be overwritten by this function
352 LCG<<<N_BLOCKS, N_THREADS>>>(cuda_random_vec, N, 42);
353
354 // ##### TASK c) #####
355 // create input and output structs on CPU and copy them over to GPU
356 SimInput_t input;
357 SimOutput_t output;
358 SimInput_t *cuda_input;
359 SimOutput_t *cuda_output;
360 cudaMalloc(&cuda_input, sizeof(SimInput_t));
361 cudaMalloc(&cuda_output, sizeof(SimOutput_t));
362
363 cudaMemcpy(cuda_input, &input, sizeof(SimInput_t), cudaMemcpyHostToDevice);
364 cudaMemcpy(cuda_output, &output, sizeof(SimOutput_t), cudaMemcpyHostToDevice);
365
366 // let a single thread do the initialization
367 initialize<<<1, 1>>>(cuda_input, cuda_output);
368
369 cudaDeviceSynchronize();
370
371 // test if initialize function worked
372 test_init<<<1, 1>>>(cuda_input, cuda_output);
373
374 cudaDeviceSynchronize();
375
376 // cudaMemcpy(&input, cuda_input, sizeof(SimInput_t), cudaMemcpyDeviceToHost);
377
378
379 init_input(&input);
380 init_output(&output, input.population_size);
381
382 Timer timer;
383 srand(0); // initialize random seed for deterministic output
384 timer.reset();
385 run_simulation(&input, &output);
386 printf("Simulation time: %g\n", timer.get());
387
388 cudaDeviceSynchronize();
389
390 //init_input(&input);
391 //init_output(&output, input.population_size);
392
393 // tried to make it work at first on CPU but segmentation fault ...
394 //run(&input, &output, random_vec);
395
396 // next step would have been to make it run with one thread on GPU
397 //run<<<1, 1>>>(cuda_input, cuda_output, cuda_random_vec);
398
399
400

```

```
401 | return EXIT_SUCCESS;  
402 | }
```