Simon Hinterseer e09925802

Exercise 09

Contents

Simon Hinterseer e09925802

1 Comparison CUDA / HIP

The code for this part can be found in the files **ex05_1_conj_grad_cuda.cpp** and **ex05_1_conj_grad_hip.cpp**.

1.1 Conversion

The conjugate gradients implementation from exercise 4 was converted to HIP. The conversion was very easy using only three steps:

- String replacement: FindReplace("cuda", "hip")
- String manipulation: every identifier, that ended in `.x` had the `.x` replaced by `_x`. The first letter is capitalized and then the substring `hip` is prefixed. E.g. `blockDim.x` -> `hipBlockDim_x`
- Kernel calls had to be converted.

1.2 Code Remarks

The original code used warp shuffles to calculate the dot product. This was replaced by a function, that uses block-level shared arrays instead. The reasoning was, that a quick internet search suggested, that while HIP is able to perform warp shuffles, potential differences in warp size and function names could lead to complications.

It should also be noted, that the original solution (I used my own solution) has an unsafe initialization of the return value of the dot product, since initialization is performed inside the kernel:

```
__global__ unsafeDotProduct(args){
	if (thread_id_global == 0)
		*global_result = 0;
	// …
	// do lots of work
	// …
	if (threadIdx.x == 0)
		atomicAdd(*global_result, local_result);
}
```

While this is unsafe, the implementation still gives convergence. There even is reason to believe, that it does not fail a single time, since the number of required iterations does not change between several runs of the program. However, the correct approach would be to initialize this variable outside the kernel. This problem was however not fixed, since the task was to compare CUDA with HIP, rather than implementing a correct dot product. This means both the CUDA and the HIP implementation have this error in the code.

1.3 Performance Comparison

The comparison shows almost perfect performance equality between the two implementations.

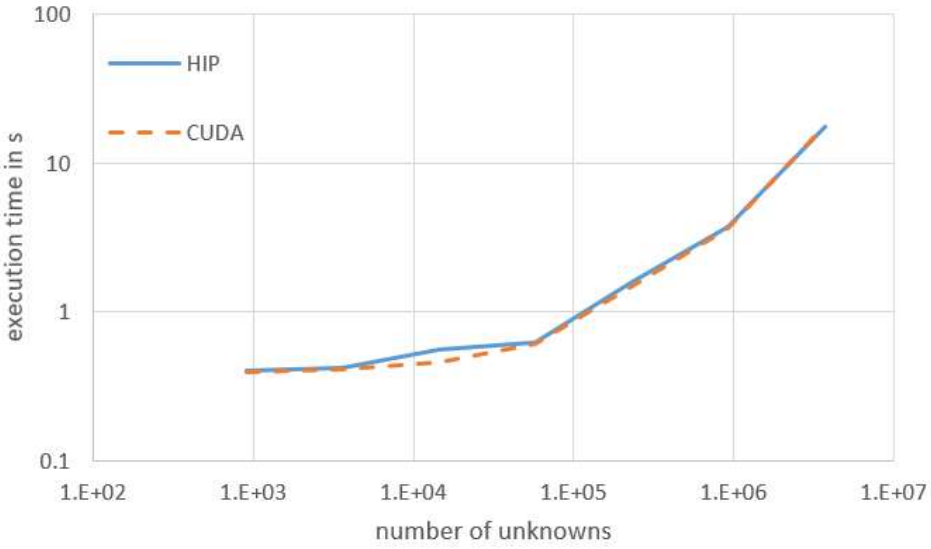

Figure 1: Performance comparison between HIP and CUDA

2 COVID Simulator

The code for a simple exemplary COVID simulation was provided. The simulation was implemented in CUDA. The code can be found in the file **covid_cuda.cpp**.

2.1 Random Numbers on GPU

Approach

The chosen approach was to generate random numbers on the GPU using a generator that combines three linear congruential generators (LCG). This was chosen over using a pre-generated pool of random numbers, since it is not easy to estimate the amount of random numbers, each thread will need when running the simulation. In addition to that, the computation power of the GPU cannot be used for creating the random numbers. The advantage would be, that the quality of random numbers would be high, since the library function `rand` is expected to use a state-of-the-art RNG.

Generator

An LCG calculates a random number like this:

$$x_{new} = (a \cdot x_{old} + c)\ mod\ m$$

Three LCGs with different periods were chosen, such that the least common multiple (LCM) of the periods is a very large number. The idea behind that is, that the largest possible period is the LCM of the three periods of the LCGs. Since no such triple of LCGs was found on the internet, some experimentation yielded these:

Table 1: Three common LCGs were used (source Wikipedia[1]). The Borland LCG was slightly modified

	numerical recipes	Borland C/C++ (modified)	ZX81
m	4,294,967,296	4,294,967,290	65,537
a	1,664,525	22,695,477	75
c	1,013,904,223	1	74

The Borland LCG's constant m was changed in order to increase the size of the LCM of the m values of the LCGs. For this triple the LCM is about $9.2 \cdot 10^{18}$, which is a lot smaller than the product ($7 \cdot 10^{24}$) but still a lot better than the period of a single LCD ($4.3 \cdot 10^{9}$).

In order to get a random number, the generator will generate three separate random numbers r_1, r_2, r_3 and return the sum modulo 1. In other words, it will return $r_1 + r_2 + r_3 - \lfloor r_1 + r_2 + r_3 \rfloor$.

Implementation Notes

An array `device_random_number` of size `GRID_SIZE*BLOCK_SIZE*3` was allocated on the GPU to hold `unsigned int` values. When generating a random number, a thread will access and update three separate random integers in `device_random_number`. The thread will access the entry with index

[1] https://en.wikipedia.org/wiki/Linear_congruential_generator

`global_thread_index*3` and the two consecutive entries. This memory access pattern is good on a CPU, but might not be ideal on a GPU (offset memory access). However, no experiments were made to improve this.

From the three integers, three floats between 0 and 1 and their sum `r = r1 + r2 + r3` are calculated. Since the call to `fmod` does not seem to be possible from a CUDA kernel, the return value is `r - int(r)`.

Functions: `myRNG, myLCG1, myLCG2, myLCG3`

The LCG constants are defined in pre-compiler statements in the first lines of the file.

2.2 CUDA implementation of the Simulator

Port of Initialization

The original program initializes in two steps – the input is initialized and the output. Since the input does not hold any large arrays, this was not moved to the GPU. Instead, it was initialized on the CPU and copied to the GPU, since the data was needed on both devices.

The large parts of the initialization were moved to the GPU. This includes the initialization of the arrays `is_infected` and `infected_on`. These arrays have the population size as length.

Furthermore, the random numbers are initialized with seeds and the two integers `num_infected_current` and `num_recovered_current` are initialized to zero for the first iteration of the loop over the simulated days.

The initialization is done in the function `init_gpu`.

Port of Simulation-Loop

The simulation loop consists of three steps:

1. determine number of infections and recoveries
2. determine a day's transmission probability and contacts based on pandemic situation
3. pass on infections within population

Only steps 1 and 3 were performed in CUDA kernels, since step 2 is computationally very lightweight.

After step 1, the numbers of actively infected and recovered people are transferred in two separate `cudaMemcpy` calls. Ideally this should have been combined to one single call, by creating a struct, that holds both values. This would avoid doubling the latency of this data transfer.

The kernel function `step1_gpu` adds thread locally the numbers of infected and recovered and uses block level shared arrays to perform a reduction. The final result is calculated using `atomicAdd`. The initialization of the values is done at the end of the kernel function `step3_gpu`.

Besides this initialization, the kernel function `step3_gpu` has only few changes compared to step 3 in the CPU version. It only parallelizes the loop over the population and calls the `myRNG` function instead of `rand`.

Execution Time comparison

The CPU Version of the Program takes about 8.1 seconds.

The GPU Version takes about 1.1 seconds. This could be improved by about 0.3 seconds, if the numbers of actively infected and recovered people would be transferred in a single call to `cudaMemcpy`.

Results comparison

Figure 2 shows, that CPU and GPU versions of the simulator yield very similar results. The differences could be caused by different random numbers being generated. It is however also possible, that the higher quality of the RNG in the CPU version causes some differences.

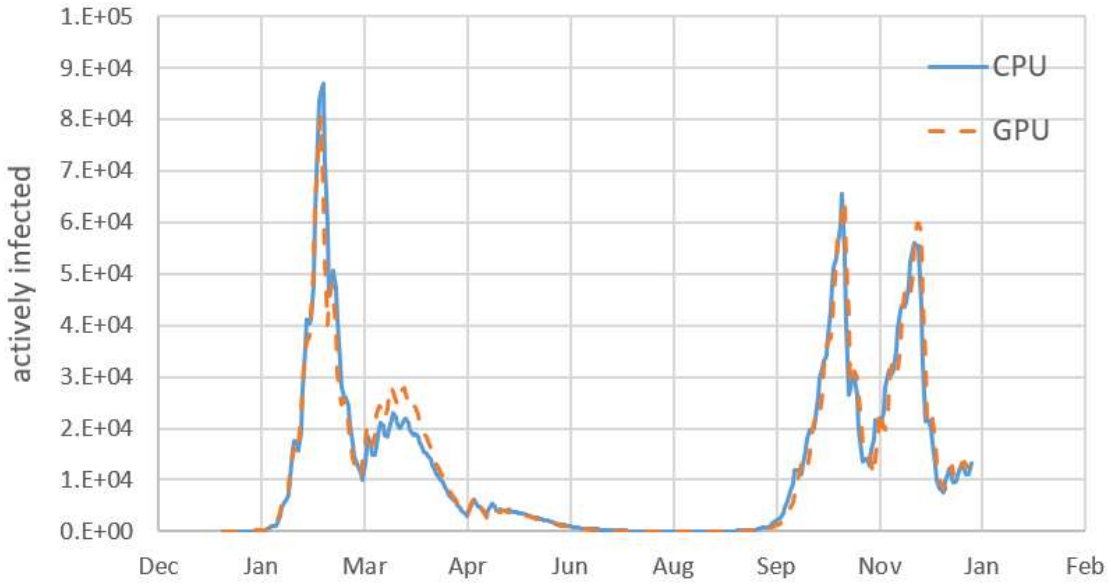

Figure 2: Number of actively infected people on CPU and GPU version of the simulator

2.3 Performance Model

Figure 3: This task was not done

2.4 Bonus: Speed up CPU Version

The CPU version of the simulation can be sped up dramatically, if the agent-based approach is dropped. The way the model is implemented at the moment does not require such an approach.

Instead, the model could just (for each day) keep track of the number of people, who were infected on that day $N_{infected}(d)$. This number defines both the number of actively infected as well as the number of immune people on later days.

The number of **actively infected** can now be calculated like this

$$N_{ainfected}(d) = \sum_{d=to}^{today-incubation\ time} \sum_{(incubation\ time+sicknes}^{duration)} N_{infected}(d)$$

The number of **immune people** is calculated from

$$N_{immune}(d) = \sum_{d=today}^{toda} \sum_{\substack{(incubation\ ti \\ sickness\ duration+ \\ immunity\ duration)}}^{(incubation\ time+sickn} N_{infected}(d)$$

This gives an immunity factor $f_{immunity} = N_{immune}/N_{total}$

Now the **infection spread** can be calculated like this:

$$N_{infected}(d) = N_{ainfected}(d) \cdot N_{contacts}(d) \cdot p_{infection}(d) \cdot \left(1 - f_{immunity}(d)\right)$$

With the parameters $N_{contacts}(d)$ denoting the daily adjusted number of contacts of that day and $p_{infection}(d)$ denoting the daily adjusted probability for the infection to spread to a contacted person.

This approach still needs to iterate over the days, but does not have to loop over the population. This is not parallelizable, because each iteration needs the result from the previous one. Hence computation on the GPU is not expected to give a speedup.

It should be mentioned, that an agent-based approach has of course much more potential for model refinement. Many of these refinements make the suggested optimization impossible.

2.5 Bonus: Model Refinement

The refined model can be run by replacing the kernel call to `step3_gpu` with a call to `step3_gpu_mod`.

The chosen model refinement was to introduce a parameter, that models how disciplined people are, when it comes to respecting a lockdown. This is implemented via an additional parameter in the `SimInpupt_t` struct:

```
double lockdown_discipline; // chance that people will respect a lockdown
```

Each time, an infected person spreads the infection, it is randomly determined, whether the ¼ reduction factor applies when determining the number of contacts. A value of 1 for `lockdown_discipline` means, it is always applied, a factor of zero means, that people will always ignore lockdowns.

Figure 4 shows, that lowering `lockdown_discipline` mostly results in longer lockdowns. This is expected, since lockdowns end, when the number of actively infected drops below a certain number. This process is prolonged, if the number of infections drops more slowly.

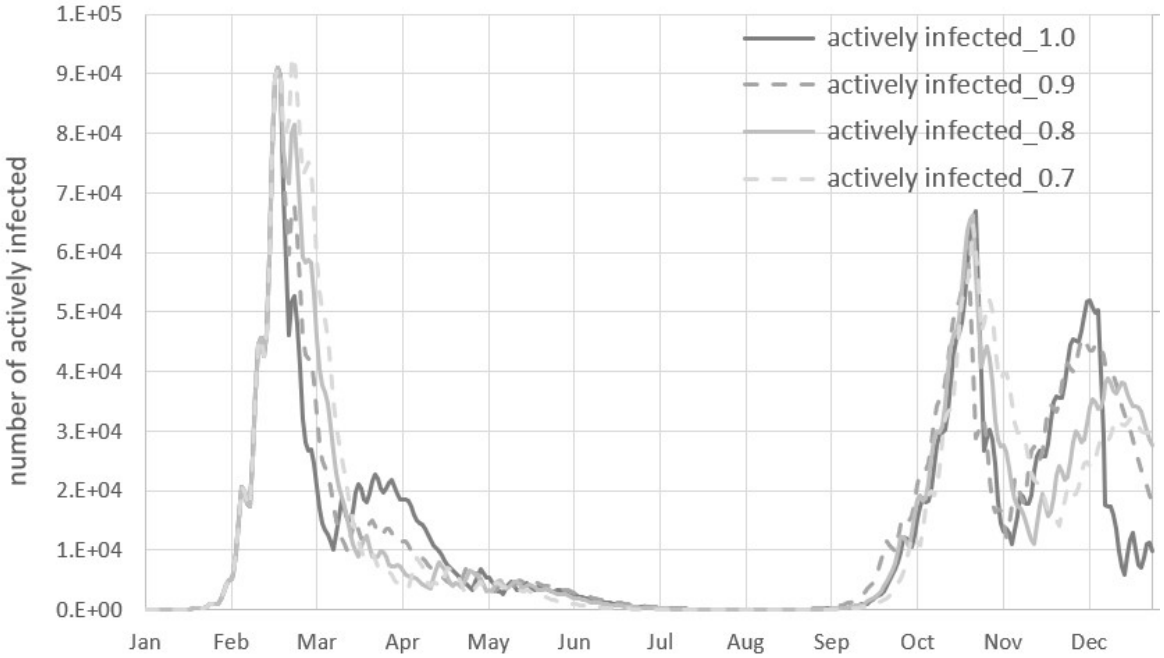

Figure 4: Infection numbers for different levels of lockdown discipline

It was observed, that when `lockdown_discipline` is set to a value of 0.6 or lower, the number of infected people will drop to zero during the summer period and the pandemic ends. This is likely due to the combination of high numbers of immune people because of increased spread of infections during lockdown phases and the reduced infection spread risk during the summer period.

In hindsight, it might have been possible to get very similar results by simply adjusting the parameter of lockdown contact reduction, that is set to ¼ per default. However, the `lockdown_discipline` parameter allows finer adjustment, since the lockdown contact reduction can only lead to one of 7 levels of contact (since infected people will always end up seeing 0-6 other people).