Exercise 09

Contents

[1 Comparison CUDA / HIP 2](#_Toc93414278)

[1.1 Conversion 2](#_Toc93414279)

[1.2 Code Remarks 2](#_Toc93414280)

[1.3 Performance Comparison 3](#_Toc93414281)

[2 2nd part 3](#_Toc93414282)

# 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**.

## 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, which was the only semi-trivial step.

## 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 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.

## Performance Comparison

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

Chart, line chart

Description automatically generated

Figure 1: Performance comparison between HIP and CUDA

# 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**.

## 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:

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, some experimentation yielded these:

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

|  |  |  |  |
| --- | --- | --- | --- |
|  | numerical recipes | modified Borland C/C++ | 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 was changed in order to increase the size of the LCM of the values of the LCGs. For this triple the LCM is about , which is a lot smaller than the   
product () but still a lot better than the period of a single LCD ().

In order to get a random number, the generator will generate three separate random numbers and return the sum modulo 1. In other words, it will return .

Implementation Notes

An array device\_random\_number of size three 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 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. 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 precompiler statements in the first lines of the file.

## 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.

Run Time comparison

Figures

## Performance Model

this task was not done 😊 sue me

no -i just didn’t know how to approach it – i would have needed to call the thing with several different population sizes??? dunno. got many points already anyways. Concentrated on other things.

## Bonus: Speed up for CPU

## Bonus: Model Refinement

Chart, line chart

Description automatically generated

Figure 2: Infection numbers for different levels of lockdown discipline

1. https://en.wikipedia.org/wiki/Linear\_congruential\_generator [↑](#footnote-ref-1)