Highly Parallel Programming of GPUs
2. Lab

Technical Notes

The installed CUDA version 12.2 does not support the old NVIDIA Quadro P400 by default. You have to force compilation for the architecture manually via:

```
nvcc --gpu-architecture=compute_61 --gpu-code=sm_61 foo.cu
```

Exercise

1. Write a program that allocates three integer vectors `a_host`, `b_host` and `c_host` with 1'000'000 elements on the host side. Initialise the elements of `a_host` and `b_host` with x`[i]=i`. Allocate corresponding device memory and initialize a and b.
   Write a kernel that computes c = a + b. Use grid-striding loops.
   After copied back check the results.

   Recommendation: Use heap memory for host side allocation (new,delete) instead of stack memory, otherwise stack might run out of memory. If you insist on stack memory, then remove the limit by „ulimit –s unlimited".

2. Write another program with **float** vectors to compute the dot product (https://en.wikipedia.org/wiki/Dot_product) c = <a, b>.
   (a[i]=i, b[i]=i).

   Recommendation: For the addition atomics are required (e.g. atomicAdd() - http://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomicadd).

Homework

Monte-Carlo Simulation

Monte-Carlo-Simulations (https://en.wikipedia.org/wiki/Monte_Carlo_method) rely on repeated random sampling over a domain, e.g. to approximate a definite integral numerically.

1. Use the curand library to create random numbers (http://docs.nvidia.com/cuda/curand/device-api-overview.html)
   - ☐ Include <curand.h> and <curand_kernel.h>
   - ☐ Allocate curandState Array (curandState for each thread and dimension)

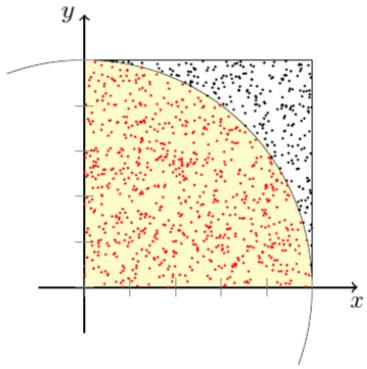
     ```
     curandState* states;
     cudaMalloc(&states, …)
     ```

   - ☐ Initialise random numbers in a separate kernel

```
//curand_init(seed, sequence, offset, &state)
curand_init(1985, idx, 0, &states[idx]);
curand_init(1985, idx+n, 0, &states[idx+n]);
// …
```

- Random numbers are obtained by curand_uniform(&state)
- Compile: nvcc -o homework02_curand homework02_curand.cu -lcurand

2. Approximate the mathematical constant Pi by a Monte-Carlo simulation (German: https://de.wikipedia.org/wiki/Monte-Carlo-Algorithmus - Probabilistische_Bestimmung_der_Zahl_Pi)



π = 3,14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209 74944 59230 78164 06286 20899 86280 34825 34211 70679 …

3. Instead of using uncorrelated random numbers you also could use evenly distributed numbers out of the properties of a CUDA kernel. Implement this approach as well.

Recommendation: For runtime analysis you can use `nvprof`,

```
nvprof ./a.out
nvprof --print-gpu-trace ./a.out
```

Random Numbers Performance #2

curand's default pseudorandom generator, XORWOW, takes quite some time to setup the random numbers. For random bit generation there exists the MTGP32 generator and the Philox_4x32_10 generator. The properties vary, check out the docs (http://docs.nvidia.com/cuda/curand/device-api-overview.html#pseudorandom-sequences). For fast random number generation you can try out the Philox_4x32_10 generator, which has fast setup times and is capable to create 4 random numbers at the same time by using functions like

```
float4 curand_uniform4(curandStatePhilox4_32_10_t *)
```

Implement the Pi Approximation algorithm with curandStatePhilox4_32_10_t and compare the runtimes. Note, that a thread now computes 4 hits in a row.