Tutorial 1: First Steps with CUDA

Siegfried Höfinger

VSC Research Center, Vienna University of Technology

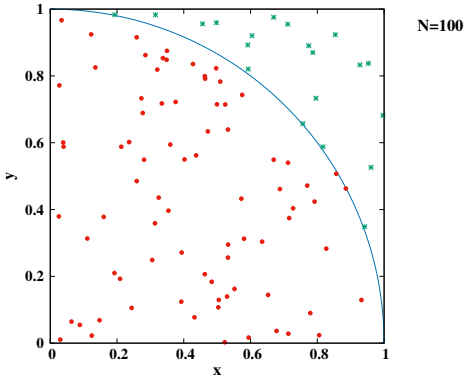
October 19, 2022

→ https://tinyurl.com/cuda4dummies/i/t/notes-t1.pdf

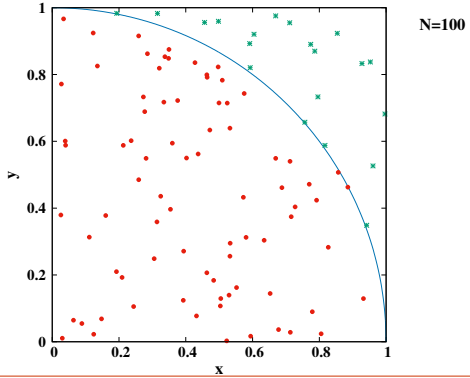
Outline

First Steps with CUDA

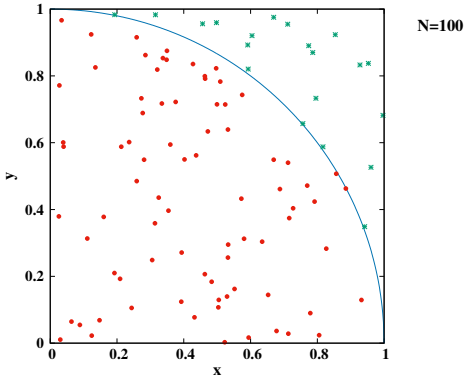
- Let us create a set of random points in the first quadrant, i.e. $x, y \in [0, 1]$



- The ratio of partial areas of the unit circle to the unit square is $\frac{r^2\pi}{4} : 1 = \frac{\pi}{4}$
- Can approximate that ratio from counting points inside the unit circle and relating them to N



- The approximated ratio $\#dots_{red}/N$ times 4 will give an approximate value for π



Exercise

Q1) *Look into pi_v0.c (the standard ANSI C version) and follow individual steps implemented there. Compile it, run it, and probably modify it. Some of the commented printf() statements can be re-activated for better understanding of various data sets used.*

→ https://tinyurl.com/cuda4dummies/i/t/pi_v0.c
10 min

A1) *Just a straightforward implementation using x[] and y[] arrays of dimension N to store random coordinates from the interval [0,1] and compute radii for each of the points and store them into array r[]. When converting r[] values to int we can directly use these resulting numbers to count all points outside the unit circle.*

Exercise

Q2) *Port the default implementation to the GPU using CUDA. The assumption is that computing radii is the most expensive step in the entire algorithm. A template, pi_template_v0.cu, can be used for a start.*

→ https://tinyurl.com/cuda4dummies/i/t/pi_template_v0.cu
15 min

A2)

i) *Writing a GPU kernel that replaces the loop where all radii, r[i], were computed*
ii) *Replace the default memory allocation with CUDA-managed unified memory*
iii) *Subsitute the loop over r[i] calculations with an appropriate kernel launch using a kernel execution configuration of appropriate size and shape*
iv) *Free the CUDA-managed unified memory in the end*

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v0.cu

Exercise

Q3) *In terms of accuracy the inital sample using $N = 100$ was not very convincing. So let's change this to higher values of N and see how the approximation of π will change (hopefully improve). Once the ANSI C version is clear, do the same thing for the CUDA version, starting with pi_template_v1.cu*

→ https://tinyurl.com/cuda4dummies/i/t/pi_v1.c
→ https://tinyurl.com/cuda4dummies/i/t/pi_template_v1.cu
20 min

First Steps with CUDA

A3)

i) *As expected increasing N will improve the quality of the approximation of π*

ii) *Only with $N = 10^8$ the first 4 digits after the decimal point become sort of converged*

iii) *Already early in the series of increasing N, we reach the maximum number of threads allowed in the threadblock, hence need to switch to the 2-level kernel execution configuration of a bunch of threadblocks on the blockgrid*

iv) *Threadblock dimensions need not necessarily fit as an integral multiple into N, so care must be taken to not refer to non-existing array elements, e.g. via padding*

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v1.cu

Exercise

Q4) *Let us take the previous solution, pi_solution_v1.cu, as new template and try to also include random number generation into the kernel code... there should be appropriate device functions around somewhere, perhaps we can make use of curand_uniform_double() ? For N = 500000000 what is the difference in kernel runtimes when carrying out measurements with 'nsys nvprof'*

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v1.cu
→ https://docs.nvidia.com/cuda/curand/device-api-overview.html
15 min

A4)

i) *The kernel code becomes much faster now (approximately 2x) despite of having to do more work, ie also computing random coordinates*

ii) *The quality of the approximation of π does not seem to be compromised, however, the impression is that resulting approximations become larger than with the CPU-based random number generator*

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v2.cu

Exercise

Q5) *Let us again take pi_solution_v2.cu as new template and try to carry out partial sums at the level of threadblocks. In particular what shall be achieved is to off-load the final step of evaluating all r[] elements already to the GPU so that when coming back from the device the host code has only to sum up a minor list of already existing partial solutions. For again $N = 500000000$ we want to know whether the overall approximation of π has remained stable, the overall execution time could be reduced and at what additional cost of kernel execution time. Do we really need the atomicAdd() ?*

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v2.cu
15 min

A5)

i) The quality of approximating π seems to be unchanged

ii) Several crucial — but time critical — changes in the kernel code have made the latter a bit slower now, $\approx +0.2$ s

iii) However, the overall benefit is largely compensating the increased kernel execution times, i.e. user time was reduced from 1.80 s to 1.61 s while system time changed from 7.55 s to 5.67 s

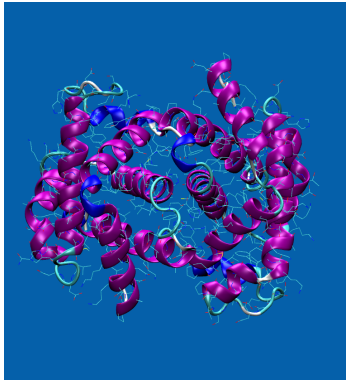
iv) Unfortunately yes, we do need the atomicAdd() as easily demonstrated from replacing it with an ordinary add operation

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v3.cu

First Steps with CUDA

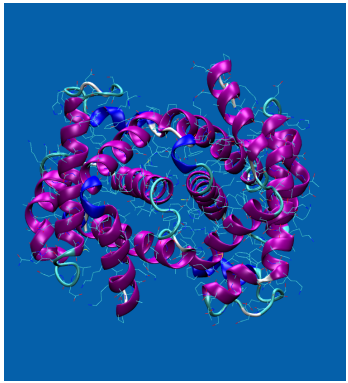
Example: Compute Coulomb Interaction for Human Oxyhaemoglobin, 1hho.pdb

- GPUs have been revolutionary to computational biology
- In molecular dynamics simulations (MD) several physical interactions are taken into account (by force fields), the most expensive of all being Coulomb interaction.



- The structure (dimer) of human oxyhemoglobin (1hho.pdb) consists of 287 residues (excluding heme), which is 4383 atoms, all of them carrying a partial charge.
- Shall try to compute the full Coulomb sum for that set of atomic charges.
- AMBER ff:

$$V_{Coulomb} = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}$$



Exercise

Q6) *Have a look at coulomb_v0.c (and perhaps the accompanying structure file MOLPDB) and examine individual steps. Compile it, run it, and probably check its logic and the resulting reference value.*

→ https://tinyurl.com/cuda4dummies/i/t/coulomb_v0.c
→ https://tinyurl.com/cuda4dummies/i/t/MOLPDB
10 min

A6) *Nothing special, just the straightforward implementation using x[], y[], z[] and q[] arrays of dimension N to store locations of atoms and corresponding partial charges. The computation of the full Coulomb sum is carried out in the standard way using a double loop over unique i,j-pairs. The resulting -230.611048407 can be taken as reference for upcoming comparisons.*

Exercise

Q7) *Port the default implementation to the GPU using CUDA. The idea is to have individual threads carry out one particular partial sum over interacting particles j and let the host then finally add up these partial sums. A template, coulomb_template_v0.cu, can be used for a start.*

→ https://tinyurl.com/cuda4dummies/i/t/coulomb_template_v0.cu
20 min

A7)

i) *In the GPU kernel each thread carries out the inner loop over j taking into account all potential interaction partners; these partial results are stored to an extra array cp[]*
ii) *Replace the default memory allocation with CUDA-managed unified memory, but pad the final sections*
iii) *Subsitute the double loop with an appropriate kernel call covering all particles*
iv) *Have the host code sum up all partial sums received from the device*
v) *Free the CUDA-managed unified memory in the end*
vi) *The result is again -230.611048407*

→ https://tinyurl.com/cuda4dummies/i/t/coulomb_solution_v0.cu

Bonus Exercise 1

QB1) *In case we do not need any of the basic data, x[], y[], r[], for any other purposes, could we just off-load all steps to the GPU and just have the host code do the final sum of all partial results ? That would be expected to be very fast because variables could become thread(block)-local... Let's take pi_solution_v3.cu as template and again work with* $N = 500000000$

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v3.cu
10 min

AB1)

i) *The quality of approximating π hasn't changed while many of the cudaMallocManaged() calls were omitted and variables used locally instead*

ii) *User time was reduced from 1.61 s to 0.09 s while system time changed from 5.67 s to 5.29 s and kernel execution time decreased from 1.5123 s to 0.0078 s*

→ https://tinyurl.com/cuda4dummies/i/t/pi_solution_v4.cu

Bonus Exercise 2

QB2) *What is the introduced error if we convert all double variables to float. Moreover, what is the greatest drawback in terms of compute performance of this current implementation ? Let's take coulomb_solution_v0.cu as template.*

→ https://tinyurl.com/cuda4dummies/i/t/coulomb_solution_v0.cu
10 min

AB2)

i) *The full Coulomb sum has changed by -0.000340753 which at first sight does not appear to be too terrible... however, it's way too much when considering that this is just the situation of an isolated small protein (lacking solvation) at a single point in time !*

ii) *A severe problem inherent in the current implementation is a very asymmetric distribution of load across individual threads in the threadblock (higher indices have significantly reduced loops over interaction partners j),*

→ https://tinyurl.com/cuda4dummies/i/t/coulomb_solution_v1.cu