ETH zürich

High Performance Computing for Science and Engineering II

Spring semester 2021

P. Koumoutsakos
ETH Zentrum, CLT F 13
CH-8092 Zürich

HW 5 - GPU programming with CUDA

Issued: May 3, 2021
Due Date: May 17, 2021, 10:00am

1-Week Milestone: Solve tasks 1 and 2

Task 1: Optimizing N-body force calculations (40 Points)

In this task you are going to optimize the computation of interaction forces as it can be found in a N-body simulation. N-body simluations allow to approximately compute the evolution of N particles (or bodies) taking into account their interaction through some given potential(s): e.g. Lennard-Jones potential in Molecular Dynamics or the gravitational potential in Astrophysics. In a naive approach one would need to compute N^2 force interactions to be able to propagate the system, but even with more sophisticated methods which have lower asymptotic complexity (e.g. particle-particle-particle mesh), the runtime will depend heavily on the speed with which we are able to compute these interactions.

a) You are now asked to optimize a kernel `computeForcesKernel` (in the file `force_kernel_0.cu`) which does exactly what has been described above, but for a toy N-body problem (of complexity N^2). The simplified equation for the total force exerted on a particle reads as follows:

$$\boldsymbol{F}_i^{tot} = \sum_{j \neq i} \boldsymbol{F}_{ji} \tag{1}$$

$$\boldsymbol{F}_{ji} = \frac{\boldsymbol{p}_j - \boldsymbol{p}_i}{|\boldsymbol{p}_j - \boldsymbol{p}_i|^3} \tag{2}$$

The kernel as it is given in the skeleton code is not yet very fast. Your task is to speed up the kernel with any improvements you consider necessary. After each successive optimization, check the computed statistics which are printed to the console: <F^2> should remain exactly the same whereas the individual components of <F> should stay in the same order of magnitude. Also you are only allowed to make changes to the file containing the kernel code. Don't forget to consult the lecture slides.

The grading will be done according to the runtime you achieve with your optimized code. To be awarded full points for this task, you will have to reach an average execution time of below 360ms (speedup of 16). In addition you are also asked to keep track and write down the successive optimizations you apply, the corresponding runtimes and achieved speedup compared to the initial kernel in `force_kernel_0.cu`.

Task 2: GPU peak performance (20 Points)

In this task we want to get more familiar with the Nvidia P100 GPU, its properties and performance characteristics.

a) **(5 Points)** Write a small program to query the GPU properties using `cudaGetDeviceProperties` or `cudaDeviceGetAttribute` from the CUDA API. Obtain the device name, its compute capability, the number of streaming multiprocessors on the device, the GPU clock rate, the memory clock rate and the memory bus width (the bus width tells you how many bit can be transferred in parallel).

The following table lists the number of single precision cores per SM for the different compute capabilities (CC):

CC	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5	3.7
Floating-point cores	8	8	8	8	32	48	192	192	192

CC	5.0	5.2	6.0	6.1	6.2	7.0	7.2	7.5	8.0
Floating-point cores	128	128	64	128	128	64	64	64	64

Note that the GPU memories (GDDR5 or HBM) are double data rate and the GPU cores can perform one fused multiply-add (FMA) per cycle. Compute the peak floating point performance (single precision) and peak memory bandwidth of the GPUs on Piz Daint from the queried data and the table above and create a roofline plot from the data (plot the hardware ceilings). Write your code in `device_properties.cu`.

b) **(15 Points)** Write a GPU kernel to attain peak compute performance using single precision data. Write your code in the skeleton file `peak_perf.cu`, which you can also use to measure the performance of your kernel. Plot your best measurement with a horizontal line in the roofline plot you have created in the previous task. We will create an anonymous ranking of your submissions and present them to you to make it a bit more competitive.

You can be creative when writing your kernel. The computed result is of secondary importance (it should not be `NaN` or `Inf`). A few things you should keep in mind when targeting peak performance:

1. Minimize memory accesses. You should reuse data from registers as much as possible.
2. You must make sure that your kernel uses fused multiply-add instructions (FMA, one addition and one multiplication per cycle). You will not reach the peak otherwise.
3. You should enable some amount of instruction level parallelism (ILP), for example by loop-unrolling.
4. Make sure you keep all the SM's busy.
5. Try to use optimization flags when compiling your code. Check `nvcc -h`.

Hint: You can generate GPU assembly code with the `--ptx` option to `nvcc`. This way you can check the instructions the compiler has generated from your code. You want to look out for FMA instructions here.

Task 3: GPU memory bandwidth (20 Points)

In this task we are discussing memory transfers associated with GPU computing. In the previous task you have determined the roofline of our target GPU for which you have determined the peak bandwidth on the device. If your application is memory bound and you have all your data *on the GPU*, this bandwidth will be your hardware limitation.

Because GPU memory is so expensive, it can often only be afforded in a small quantities. What if you are working with applications that require a lot of memory (e.g. fluid dynamics simulations)? In such cases you would rather prefer to use the host memory, as it is usually 4 to 8 times larger (on Piz Daint it is 4). An application that runs on the host and uses the GPU as an *accelerator* is called heterogeneous. CPU and GPU vendors are in competition which means that there must be an interface to communicate between CPU and GPU. This interface can be a major bottleneck in heterogeneous computing and we are therefore interested in the achievable bandwidths through this interface.

a) **(5 Points)** What is the name of the interface that connects the GPUs on Piz Daint with the host CPU? What is the maximum bandwidth that you can expect?

b) **(15 Points)** Similar to cache lines, DRAM memory has a concept of *pages* to improve the performance of memory accesses. Whenever you allocate memory in your code it will be *pageable* memory. The CUDA driver can not access pageable memory directly and needs to allocate a *page-locked* buffer first before any transfer can happen. This has implications on performance for data transfer if pageable buffers are used. The CUDA API provides functions for memory allocation on the host that allocates page-locked memory and therefore avoids the additional CUDA driver overhead (see `cudaMallocHost`). To allocate memory on the GPU you can use `cudaMalloc`. Copy operations in CUDA are performed with `cudaMemcpy` where the function requires an argument that defines the direction of the copy (there are more but they are not relevant for this exercise):

   1. `cudaMemcpyHostToDevice` copies from host memory to device memory
   2. `cudaMemcpyDeviceToHost` copies from device memory to host memory
   3. `cudaMemcpyDeviceToDevice` copies memory between two locations on the device

   We want to measure the bandwidth of copy operations from host to device (H2D), device to host (D2H) and device to device (D2D). Implement the `transfer` function in the `memory_copy.cu` skeleton code to perform measurements. Your implementation must take into account the direction of the copy (indicated by the argument `dir`) and whether the host allocation should be pageable or page-locked (argument `method`).

   Use your implementation to collect data for the following two measurements:

   1. Pageable host memory
   2. Page-locked host memory

   For each measurement, generate a log-log plot of the collected data (H2D, D2H, D2D) with the copy volume on the abscissa (in MB) and the measured bandwidth on the ordinate (MB/s). Comment on your observations.

Guidelines for reports submissions:

- Submit **a single zip file** with your solution (code in a compressed format) via Moodle until May 17, 2021, 10:00am. The total size of the zip file must be **less than 20 MB**.