ETH zürich

High Performance Computing for Science and Engineering II

Spring semester 2022

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

HW 5 - GPU programming with CUDA

Issued: May 2, 2022
Due Date: May 16, 2022, 10:00am

1-Week Milestone: Solve task 1

Task 1: Implementing DGEMM (40 Points)

In this question you will implement a double precision version of GEMM (*GEneral Matrix Multiplication*):

$$\mathbf{C} \longleftarrow \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C} \tag{1}$$

with $\mathbf{A}(m \times k), \mathbf{B}(k \times n)$ and $\mathbf{C}(m \times n)$. GEMM is a simple algorithm you would have learned in Linear Algebra 1, yet it is fundamental in many scientific applications. It is one of a handful of algorithms where the GPU can achieve peak performance, and although such an implementation is out of the scope of this exercise, it is a good starting point of exposure to some of the core concepts that make up a successful CUDA kernel.

a) **(6 Points)** We begin with a small warm-up to become more familiar with the Nvidia P100 GPU on Piz Daint. Clone the CUDA Samples repository to your home folder. Determine the version of CUDA Toolkit available to you, and checkout the repository you just cloned at an appropriate tag. This repository contains sample codes which you can use as a reference in the future.

   Now enter `cuda-samples/Samples/deviceQuery`, make and launch. Using the information provided by this program about the Nvidia P100, deduce the double precision peak performance, and maximum device RAM (global memory) bandwidth. Note that the GPU memories (GDDR5 or HBM) are double data rate, meaning that data is transmitted twice per clock cycle. You may consult this table to deduce the throughput of native arithmetic instructions.

b) **(6 Points)** For $m = 2^{13}, k = 2^{12}$ and $n = 2^{12}$ use the Roofline model to determine whether DGEMM is memory or compute bound, and use this result for attainable performance to predict its runtime on a Tesla P100 assuming all matrices already reside on device memory.

c) **(28 Points)** In `my_dgemm_1.cu` you will find a naive implementation of DGEMM, where each thread computes an element of $\mathbf{C}$. This implementation achieves just above 1% of peak performance on the Nvidia P100. For comparison, cuBLAS's implementation achieves 96% of peak performance.

   In this part of the question you will attempt to make things better, and implement a DGEMM kernel which uses shared memory. You can consult the Shared Memory section of the "CUDA C Programming Guide" to understand the idea, and algorithm behind blocked matrix multiplication. The matrices are stored in *column major* order, and for the sake of simplicity you may assume that m, n, k are multiples of 256.

To write your kernel follow the template in `my_dgemm_1.cu`. For correct linking to `main.cu`, you have to wrap your implementation in a `myDgemm` function, and define the launch configuration for your kernel there. You can create new files using the naming convention `my_dgemm_*.cu` and append the `Makefile` at:

```
all: my_dgemm_1.exe my_dgemm_2.exe
```

to include them in the build rules. To confirm the correctness of your kernel you can refer to the root mean square error with respect to the cuBLAS implementation of DGEMM. To achieve full points in this question you need to beat $215\,\mathrm{ms}$ for the problem size defined in part (b).

d) **(optional)** Using shared memory only achieves about 28% of peak performance. To do better, you need to take advantage of registers available to each thread. Unfortunately, as a programmer you cannot explicitly allocate variables in registers, but you can leverage them through local variables. Local variables are variables which are local/private to each thread. Memory is allocated for them either in registers or global memory, hence, access to local variables may be latency free at best, or as expensive as access to global memory at worst. Sufficiently small allocations which do not exceed available register space per thread with access patterns that can be determined at compile time are likely to end up in register space. To verify whether a local variable was allocated in registers or global memory you can check the PTX instructions generated for your kernel.

You can read this technical blog to learn more about using registers in GEMM and use some of those ideas if you would like to improve your kernels. Note that completing this exercise may be a very time consuming process, and not essential in the scope of this course. Nevertheless, you're encouraged to read through the solution codes to gain exposure to the process behind writing efficient CUDA kernels.

Task 2: 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 an N-body simulation. N-body simulations 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.

You are 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:

$$\boldsymbol{F}_i^{tot} = \sum_{j \neq i} \boldsymbol{F}_{ji} \tag{2}$$

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

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 $360\,\mathrm{ms}$ (speedup of 16). You are encouraged to keep track and implement the successive optimizations you apply in separate files similarly to the previous question. The following is the list of optimizations you should try to apply:

- removing a diverging `if` branch
- use faster memory for recurring accesses
- use the math intrinsics where possible

Guidelines for reports submissions:

- Submit **a single zip file** with your solution (code in a compressed format) via Moodle until May 16, 2022, 10:00am. The total size of the zip file must be **less than 20 MB**.
- There are 80 available points in this homework. To get a grade of 6/6 you need to collect 60 points. Collecting 50 points results in a grade of 5/6 and so on.