ETHzürich

High Performance Computing for Science and Engineering II

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

Spring semester 2020

# HW 5 - CUDA Basics and Optimizations

Issued: April 27, 2020
Due Date: May 11, 2020, 08:00am

**1-Week Milestone:** Solve task 1 and task 2.

The skeleton codes for this homework are located in the gitlab repository: https://gitlab.ethz.ch/hpcse20/exercise.

## Task 1: CUDA Benchmarks

In this exercise you will perform a series of small benchmarks to get familiar with GPUs and CUDA[1]. You will be asked to measure the execution time of launching a kernel, of copying data to GPU, of accessing memory in various patterns, and of performing certain computation.

To measure the execution time of some function `func()`, invoke the function $N \gg 1$ times and compute the average execution time. Choose $N$ such that the total execution time is in the order of 0.1–1s. Use the following measuring procedure:

1. run the function `func()` $\lfloor 0.1N \rfloor + 1$ times as a warmup,
2. synchronize the device,
3. record the current time $t_0$,
4. run the function `func()` $N$ times,
5. synchronize the device,
6. record the current time $t_1$,
7. report $\Delta t = (t_1 - t_0)/N$.

*Hint:* Consider implementing a template function
`template <typename Func> double benchmark(int N, Func func);`
which performs the described measuring precedure and returns $\Delta t$ in seconds.

a) [20pts] **Overhead of launching a kernel.**
   1. [5pts] Create an empty kernel `emptyKernel()` which takes no arguments and does nothing. Measure the time it takes to launch and execute the kernel with 1 block and 1 thread per block.
   2. [3pts] After each launch, synchronize the device. What is the time per call now?

[1]If you do not have a Piz Daint account yet, please send an email to kicici@ethz.ch.

3. [7pts] Benchmark `emptyKernel()` with the following numbers of blocks $B$ and threads per block $T$:
   (a) $B = 1, T = 1$,
   (b) $B = 1, T = 32$,
   (c) $B = 1, T = 1024$,
   (d) $B = 32, T = 1024$,
   (e) $B = 1024, T = 32$,
   (f) $B = 32768, T = 1$,
   (g) $B = 32768, T = 32$.
   (h) $B = 32768, T = 1024$.

   Benchmark each case separately, with synchronization between launches enabled. What execution times do you get? Explain the results. Why are some cases equally fast even with different $B \times T$, and why do some cases with equal $B \times T$ differ drastically?
4. [5pts] Measure the time it takes to run an empty OpenMP parallel region with 12 threads. Run both benchmarks on Piz Daint hybrid nodes. Compare the results with launching a CUDA kernel.

b) [30pts] **Memory.**

Measure the performance of various memory-related operations, with respect to the total size of buffers $K$, where $K$ varies exponentially from $17$ to $50'000'001$ (see code). Since the execution time increases with $K$, make $N$ dependent on $K$, such that the each case runs for approximately the same time.

1. [5pts] Implement allocation and deallocation of buffers in the function `subtask_b` in the file `benchmarks_b`.
2. [5pts] Measure the execution time of synchronously copying $K$ `doubles` from the host to the device. Report your measured bandwidth in GB/s.
3. [10pts] Write and benchmark a kernel that copies a permutation of one array of $K$ `doubles` to another: $a_i \leftarrow b_{p_i}$. The skeleton code includes the following cases of $p_i$:
   (a) $p_i = i$,
   (b) $p_i = (2i)\%K$,
   (c) $p_i = (4i)\%K$,
   (d) $p_i = i$ initially, then $p$ is split into parts of length 32, and each part is permuted,
   (e) $p$ is a random permutation of $\{0, 1, \ldots, K-1\}$.

   Benchmark each case and explain the differences in performance.
   *Optional:* Test not only $a_i \leftarrow b_{p_i}$, but also $a_{p_i} \leftarrow b_i$. Note that the patterns and the odd values of $K$ were chosen such to avoid race condition.[2]
4. [5pts] Write and benchmark a kernel that performs a vector addition $a \leftarrow a + b$, where $a$ and $b$ are vectors of $K$ `doubles`. Report GFLOP/s. For what $K$ do you get highest performance? What does this $K$ correspond to?
5. [5pts] Make a kernel that repeats the operation $a_i \leftarrow a_i + b_i$ 100 times. Report GFLOP/s. Compare results with the previous case.

[2]Optional tasks in this homework provide no bonus points.

c) [20pts] Here we demonstrate the computational power of GPUs by approximating $\pi$ using the first $K = 2^{30} \approx 10^9$ terms of the Leibniz formula:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}. \tag{1}$$

Write a kernel that sums this series (each thread sums only a part of the series). Store the result in a `double` array, copy it to the host and accumulate all partial results. Make sure your result is correct. Experiment with the number of blocks $B$ and number of threads per block $T$. Try to achieve the best performance possible. Does changing $B$ and $T$ still affect the performance as it did in the subtask 1a?

Compare the performance with a simple `#pragma omp parallel for` implementation.

d) [5pts] Imagine we have a CUDA kernel $f(M)$ whose evaluation time is proportional to $M$. We want to evaluate the kernel $f$ for $K = 10^6$ different values of $M$. We sample each $M_1, M_2, \ldots, M_K$ uniformly from a range $[1, 10^6]$, and measure it takes $5.23$ seconds to execute the kernel (one evaluation of $f$ per thread). Then, we sort all values $M_k$, run again and now measure only $2.67$ seconds. What happened, why did the order affect the performance? Why the factor of $2$?

## Task 2: N-body code performance optimization

You are given a CUDA code that computes total forces exerted on bodies in a 3D gravitational N-body system. The total force $\mathbf{F}_i^{\text{tot}}$ on the body $i$ is given by the following simplified equations:

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

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

where $\mathbf{p}_i$ is the position of the body $i$.

Forces $\mathbf{F}_i^{\text{tot}}$ are computed by a CUDA function `computeForcesKernel` in `nbody_0.cu`. The function is executed once for each $i \in \{0, ..., N-1\}$, and it iterates over all $j$ to compute and accumulate forces $F_{ji}$. Your task is to improve the performance of this kernel by applying few consecutive optimization techniques. *Write down the execution time of the initial code, and of each of the optimized versions.*

a) [5pts] The for-loop in the initial code reads and writes to the array `f` $N$ times, which greatly degrades the performance. Find a way that writes to the array `f` only once per thread.

   *Note:* The reason why the compiler does not do this optimization automatically is because it cannot assume that writing to `f` does not modify the array `p`.

   *Optional #1:* Instead of manually resolving this *aliasing*, inform the compiler that the memory accessible through pointers `p` and `f` does not overlap by decorating them with the `__restrict__` attribute. Do you get the same performance boost? See CUDA optimization hints for more information.

   *Optional #2:* Go back to Question 1b, Subtask 5 and apply the same optimization.

b) [5pts] The kernel code repeats the same calculation multiple times. Store these intermediate results in variables and reuse them. Why didn't the compiler do this optimization automatically?

c) [5pts] The force computation contains two expensive operations: division and square root. Optimize the computation by using the built-in CUDA `rsqrt` function, where `rsqrt(x)` returns $1/\sqrt{x}$. [3]

d) [20pts] Utilize shared memory to reduce the amount of accesses to DRAM. Ensure that each block reads the array `p` from the DRAM only once.

   *Note:* Expect about 10% performance boost here.

*Recommendations:* Start with the implementation in `nbody_0.cu` and save your optimized versions in `nbody_a.cu`, `nbody_b.cu`, `nbody_c.cu` and `nbody_d.cu`, one for each subquestion respectively. Check the generated `Statistics` output to verify if the computed forces are still correct after your changes.

[3]For more information, see CUDA C++ Programming Guide, Appendix E.1, Table 7.

## Task 3: Electrostatic potential and the Jacobi method

Jacobi method[4] is an iterative algorithm for solving a linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{4}$$

where $\mathbf{A}$ is a known square matrix, $\mathbf{b}$ a known vector, and $\mathbf{x}$ the solution vector we want to compute. The method finds $\mathbf{x}$ by starting from an initial guess $\mathbf{x}^{(0)}$ and iteratively refining the solution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, N. \tag{5}$$

We will use the Jacobi method to solve a 2D electrostatics problem

$$\frac{\mathrm{d}^2\varphi}{\mathrm{d}x^2} + \frac{\mathrm{d}^2\varphi}{\mathrm{d}y^2} = -\rho, \quad (x, y) \in [0, L]^2, \tag{6}$$

$$\varphi(x = 0, y) = \varphi(x = L, y) = \varphi(x, y = 0) = \varphi(x, y = L) = 0,$$

where $\rho$ is a given charge distribution (note the minus sign) and $\varphi$ the unknown potential. The equation 6 can be represented as a linear problem 4 by discretizing the space and the differential operators, and by introducing 2D indices $i = (i_x, i_y)$:

$$\frac{1}{h^2} \left( \varphi_{i_x+1,i_y} + \varphi_{i_x-1,i_y} + \varphi_{i_x,i_y+1} + \varphi_{i_x,i_y-1} - 4\varphi_{i_x,i_y} \right) = -\rho_{i_x,i_y}, \quad i_x, i_y = 1, 2, \ldots, N,$$

$$\Downarrow$$

$$A_{i_x,i_y,j_x,j_y} = \frac{1}{h^2} \left( \delta_{i_x,j_x+1}\delta_{i_y,j_y} + \delta_{i_x,j_x-1}\delta_{i_y,j_y} + \delta_{i_x,j_x}\delta_{i_y,j_y+1} + \delta_{i_x,j_x}\delta_{i_y,j_y-1} - 4\delta_{i_x,j_x}\delta_{i_y,j_y} \right) \tag{7}$$

$$\sum_{j_x,j_y} A_{i_x,i_y,j_x,j_y} \varphi_{j_x,j_y} = -\rho_{i_x,i_y}$$

where $N$ is the number of discretization points per axis, $h = L/N$ the cell size, and $\delta_{ab}$ the Kronecker delta symbol.

Your task is to implement the algorithm 5 for the matrix $\mathbf{A}$ defined in the equation 7.

a) [10pts] Identify non-zero diagonal and non-diagonal terms of $A_{ij} = A_{i_x,i_y,j_x,j_y}$ and write down the equation 5 with double indices $i = (i_x, i_y)$ and $j = (j_x, j_y)$.

b) [20pts] Implement a CUDA function `jacobiStep` that takes the vectors $\rho$ and $\varphi^{(k)}$ as input as computes $\varphi^{(k+1)}$ (see the code for details). Note that the matrix $\mathbf{A}$ never has to be stored in memory! As a boundary condition, set $\varphi_{i_x,i_y} = 0$ for all boundary cells ($i_x = 0 \vee i_x = N-1 \vee i_y = 0 \vee i_y = N-1$).
*Note:* You can visualize the data with the provided Python script `visualize.py`. To make it work, copy $\varphi^{(k)}$ from the GPU and pass to the function `dumpCSV` (see the code).

c) [10pts] Implement the iterative algorithm that calls `jacobiStep` `numIterations` times. Use only 2 buffers for all $\varphi^{(k)}$, and alternate which represents $\varphi^{(k)}$, which $\varphi^{(k+1)}$.

d) [20pts] Implement a CUDA function `computeAphi` that computes $\mathbf{A}\varphi$ for given $\varphi$. Download the result value of $\mathbf{A}\varphi$ and pass it to the function `printL1L2` to print the L1 and L2-norm of the error $\mathbf{A}\varphi - (-\rho)$. If the algorithm is correctly implemented, the norms should decrease with more iterations.

[4] https://en.wikipedia.org/wiki/Jacobi_method

**Guidelines for reports submissions:**

- Report all your answers in a pdf. Archive your pdf and source code (e.g.: .tar, .zip) and submit it via Moodle until May 11, 2020, 08:00am.