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

HW 6 - GPU programming with CUDA (cont.)

Issued: May 16, 2022
Due Date: May 30, 2022, 10:00am

1-Week Milestone: Solve task 1

Task 1: Reduction (25 Points)

Let $a_0, a_1, \ldots, a_{N-1}$ be an array of `doubles` for some $N \in \mathbb{N}$. Your task is to perform a reduction operation

$$r = a_0 \oplus a_1 \oplus \cdots \oplus a_{N-1},$$

where $\oplus$ is some *associative* operator[1]. In this task, we consider two operators: $x \oplus y = x + y$ and $x \oplus y = \max(x, y)$.

We start with a warp-level reduction ($N = 32$), continue with block-level reduction ($N = 1024$) and grid-level reduction with $N \leq 1024^2$, and finish with arbitrarily large values of N.

a) **(3 Points)** Implement a device function `double sumWarp(double a)` that returns the sum of all values a within a single warp. The function must return the correct result on at least the 0th thread of each warp (other threads may return arbitrary values). Use the minimum number of required `__shfl_*_sync` operations.

Optional: Think how to do a warp-level reduce-all operation, where every thread returns the total sum, while keeping the same performance.

b) **(6 Points)** Implement a device function `Pair argMaxWarp(double a)` that returns the maximum value a within a warp, together with the position of the maximum (a number between 0 and 31, inclusive). Here, `Pair` is a struct containing a `double max` and an `int idx`. The function must return the correct result on (at least) the 0th thread of each warp. You may assume all values a are distinct.

c) **(6 Points)** Pick one of the following:

- Implement a device function `double sumBlock(double a)` that returns the sum of all values a within a single block of size exactly $N = 1024 = 32^2$.
- Implement a device function `Pair argMaxBlock(double a)` that returns the maximum and the location of the maximum of all values a within a single block of size exactly $N = 1024 = 32^2$. The location is now a number between 0 and 1023.

It is sufficient for the 0th thread of a block to return the correct result. *Hint:* Utilize the functions from previous subquestions. Think what mechanism you can use to exchange information between threads (or warps) of a *single* block.

[1] If $\oplus$ were not associative, we would not be able to utilize parallelism.

d) **(7 Points)** For more than 1024 elements we would like to parallelize the reduction using multiple blocks[2]. Utilize the single-block reduction from the previous subquestion to implement multi-block reduction for $N \leq 1024^2$.

e) **(3 Points)** Propose at least one way to perform sum reduction for a hypothetical case of $N = 2 \times 10^9$, without copying any data back to the host.

[2]We could always implement reduction for an arbitrary N even with a single thread, but then we would not have any parallelism.

Task 2: Inclusive Scan (25 Points)

Inclusive scan (*inscan*) is an operation that takes as input an array of size N

$$\{a_0, a_1, a_2, \ldots, a_{N-1}\}$$

and produces an array of partial reductions

$$\{a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus \cdots \oplus a_{N-1}\},$$

where $\oplus$ is a given associative binary operator.

For example, for the following array of size $N = 6$:

$$\{5, 1, 3, 2, 3, 2\}$$

and summation operator $+$ as the operator $\oplus$, the expected result is

$$\{5, 6, 9, 11, 14, 16\}.$$

The goal of this task is to implement the inclusive scan algorithm in increasing order of complexity from smaller to larger maximum array sizes N. Here, we focus only on summation inscans. Refer to the slides for the algorithm details (pick either of the two algorithms).

a) **(8 Points)** Implement the function `Scan::inclusiveSum` in `src/scan.cu` such that it supports array sizes N of up to 32. Test the code by running `./inscan --warp`. Hint: consider implementing a standalone device function that takes one value as an argument and returns the inscan of values in the current warp.

b) **(8 Points)** Update the `Scan::includeSum` function to support up to $N \leq 1024$ elements. For simplicity, use 1024 threads per block.
Test your code with `./inscan --block`.

c) **(9 Points)** Implement the inscan for $N \leq 1024^2$ as described in the slides.
Test and benchmark your code with `./inscan --medium`. Report the execution times.

d) (optional) Propose how to extend the algorithm #1 from slides to $N > 1024^2$. Implement either algorithm #1 or #2 for sizes $N > 1024^2$ and test it with `./inscan --large`. Run the code with `nvprof --print-gpu-trace ./inscan --large --profile` to profile your and the CUB implementation. The `--profile` flags instructs the code to run only one scan. Compare the performance of your kernels with the ones from CUB.

Task 3: Cell Lists (30 Points)

In this task we look into optimizing N-body force computation using cell lists. We have N particles with coordinates $\mathbf{r}_i \in \mathbb{R}^2$ for $i \in \{0, \ldots, N-1\}$. For a pair of particles (i, j), we define a repulsive short-range force $\mathbf{F}_{ji}$ that the particle j exerts on the particle i:

$$\mathbf{F}_{ji} = \begin{cases} \dfrac{-\mathbf{r}_{ji}}{(\alpha + r_{ji}^2)^{5/2}} & \text{if } r_{ji} < r_c \\ \mathbf{0} & \text{if } r_{ji} \geq r_c, \end{cases} \tag{1}$$

where $\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i$, r_c is the cutoff radius and $\alpha > 0$ a fixed parameter.

The total force $\mathbf{F}_i$ exerted on the particle i is given by

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ji}. \tag{2}$$

a) **(15 Points)** Implement the `CellList::build` function in `cell_list.cu` that builds the cell lists by (1) computing the cell sizes, i.e. the number of particles contained in each cell, (2) computing the cell offsets, i.e. starting locations of each cell in the sorted array of particles, and (3) copying the particles to the sorted array, with respect to the cell they belong to. To compute the cell offsets, you can use the inscan operation provided in `scan.h`.

b) **(15 Points)** Implement the force kernel in `interaction.cu` that utilizes cell lists to efficiently determine which particles are within the cutoff radius of each other. For each particle i, compute the force contribution only from the particles j that either belong to the same cell or to one of the 8 neighboring cells. The force F_{ji} is provided in `interaction.h` and a reference $\mathcal{O}(N^2)$ code in `brute_force.cu`. Test your code by running `./cell_list`.

Guidelines for reports submissions:

- Submit **a single zip file** with your solution (code in a compressed format) via Moodle until May 30, 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.