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

# HW 6 - CUDA Reduction, SSA and asynchronous commands

Issued: May 11, 2020
Due Date: May 25, 2020, 08:00am

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

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

## Task 1: Reduction

You are given an array $a_0, a_1, \ldots, a_{N-1}$ 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 *commutative* 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) [10pts] 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) [10pts] 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.

c) [15pts] 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 $N = 1024 = 32^2$.

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

- 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.

d) [20pts] 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) [5pts] Explain 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. Note that the maximum number of blocks in one kernel launch is $65536$.

[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: SSA - Trajectory Binning

In this task we are revisiting the Stochastic Simulation Algorithm (SSA) of Homework 4 Task 1. Please review the exercise description and the solution code from HW4 Task 1.

Here, you are asked to implement the GPU kernels for the **Trajectory binning**. A trajectory from SSA shows one possible realization of the time evolution of the system. Trajectory binning is a simple way to generate average trajectories over a number of sample trajectories. Since we do not have the same time steps for all samples (i.e. array of triples (Sa, Sb, t)), we need to average over time and over the samples. We assume that the simulation is run for times $t \in [0, t_{end}]$. We can then compute averaged quantities $\overline{S_{ik}}$ for realizations of $S_i$ in bin $k$ associated to the time interval $[k\delta t, (k+1)\delta t)$, whereas $\delta t$ is the width of the bins and $k = 0 \dots \lceil t_{end}/\delta t \rceil - 1$.

a) [10pts] In the CUDA version of the SSA algorithm (see `dimerizationKernel`) each thread simulates one trajectory for `numIters` time steps. Some trajectories may reach $t_{end}$ and finish, some may not. Therefore we need to check if all trajectories finished, and if not, we call the `dimerizationKernel` again, until all trajectories reached $t_{end}$.

   Implement the CUDA kernel `reduceIsDoneKernel`. This function stores the amount of samples/trajectories which are finished within a block and stores the count in the array `blocksDoneCount`.

b) [20pts] Implement the binning functionality described above. For this, sum the quantities of $Sa$ and $Sb$ associated to a bin, e.g. $\sum_{i:t(i)\in[k\delta t,(k+1)\delta t]} Sa(i)$, and store the accumulated value in a buffer. Try to parallelize the binning mechanism as much as possible using multiple threads and blocks. Also store the number of summation terms in a buffer such that you can average the calculated values later. You may want to make use of the reduction kernels implemented in Task 1 of this Homework.

c) [10pts] Implement a kernel in order to average the bins calculated in the previous subtask. Make sure that the implementation is correct (compare with the results from Homework 4). Finally, deallocate all extra buffers you allocated.

Please see the `README` file for compilation instructions. For your implementation, try to follow the `TODO` comments in the code. The files that need to be modified are: `ssa.cu` (allocate and deallocate memory, add function calls of your kernels), `kernels.cu` (for the actual implementation of the kernels) and `kernels.h` (extend with your function prototypes).

## Task 3: Communication–computation overlap

Imagine we are operating on so large buffers that they do not even fit in the GPU memory[3]. For example, on Piz Daint, nodes have 64GB of host memory, while GPUs have only 16GB, so if we require 50GB of data, the data has to reside on the host.

If we need to apply a computationally expensive kernel on these buffers, using a GPU may be beneficial. However, for the data to fit on the GPU, it has to transferred and processed in chunks. Your task is to implement the function `runAsync`, which executes the given kernel on an array of size $N$, in chunks of size $C$, using $S$ CUDA streams. By using more than one stream, we hope to parallelize communication and computation.

The skeleton code includes two dummy kernels `fastKernel` and `slowKernel`, which emulate a memory-bound and a compute-bound kernel, respectively. The kernels take an input buffer $a$, an output buffer $b$ and the chunk size.

a) [10pts] Create the streams, and allocate chunk memory for each of the streams. Each stream requires two chunk buffers: one for $a$, one for $b$. Do not forget to destroy the streams and free the memory at the end.

b) [20pts] Split the array `aHost` logically into chunks of size $C$. For each chunk, asynchronously copy it from the host to the device, (asynchronously) launch the kernel, and asynchronously copy the result back to the host. In the case $N$ is not divisible by $C$, truncate the last chunk.

   The skeleton code will check if the whole array was processed. Do not forget to synchronize the streams before the check.

   Report the speedup compared to the synchronous execution `runSync` which uploads and processes everything in one go (as a simplification, we work with buffers that do fit in the GPU's memory).

c) [10pts] Use `nvprof -o output.%h.%p.nvvp ./overlap` to profile your code, and `nvvp` to visualize the result. You will need to pass the `-Y` flag to `ssh` (both for `ela` and `daint`) to use `nvvp`. Take a screenshot of the visualization for the slow and the fast kernel. How many different operations can the GPU execute in parallel? How does that depend on their type (copying vs kernel)?

   Note: To make the visualization cleaner, instead of invoking `runBenchmarks()` from `main()`, invoke `profile()`.

[3] Such scenarios are plausible in computational fluid dynamics.

**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 25, 2020, 08:00am.