Test and Practice 3 - Synchronization and Performance

For this TP, a **report** of txt format is to be written and uploaded with the codes. Codes should be clean, concise and commented. Avoid anything unrelated to the computation. All the necessary files can be downloaded from e-learning. The CUDA runtime API reference (the doc) is here.

Exercise 1: Why not working?
Compile and run `chaos1.cu` and `chaos2.cu`. They are not working as intended. Correct both pieces of codes (saved in the original file) and explain.

Exercise 2: We don't have time!
Compile and run `search.cu`, in which we search for 42 in an array using four methods.

1. Explain how each method works.
2. Compare the performance of all methods under different situations. Detail and explain your observation.

Problem 1: Pagerank
Pagerank is an algorithm that compute the relative importance between nodes in a directed graph. A more advanced and tweaked version of Pagerank is used by Google to rank the importance of webpages. We now implement the Pagerank algorithm on a restricted type of directed graph.

Imagine that we have a certain number of webpages (given by the constant `COUNT`), indexed from `0` to `COUNT-1`, each has a certain number of hyperlinks (given by the constant `LINK_PER_PAGE`). Each hyperlink is represented by some `uint2 l`. If it is on the webpage `i`, pointing to the webpage `j`, then `l.x = i` and `l.y = j` (we also write $i \to j$).

Informally speaking, the pagerank of a webpage can be considered as the probability that a virtual "Internet surfer" visits this webpage at some moment. Such a surfer starts uniformly randomly from any webpage. Upon the visit of a webpage, it has a

certain probability (called the *damping factor*, the constant `DAMPING`) to follow one of the hyperlinks to another page with equal probability. Otherwise, it will jump to a random page. When the surfer has been visiting webpages for a long time, the time it spends on any webpage will converge, and the limit is the pagerank of each webpage.

Given a set of hyperlinks, we can compute the pageranks of each involved webpage using an **iterative** method. We start by a uniform pagerank vector `float* pr`, with each `pr[i]` equals to `1/COUNT`. For each time we have a pagerank vector `float* oldp`, we compute a new pagerank vector `float* newp` using the equation

$$\texttt{newp[j]} = \frac{1 - \texttt{DAMPING}}{\texttt{COUNT}} + \texttt{DAMPING} * \sum_{\texttt{i},\texttt{i}\to\texttt{j}} \frac{\texttt{oldp[i]}}{\texttt{LINK\_PER\_PAGE}}.$$

Here, the first term comes from random jumps of the surfer, and the second comes from the surfer following some link from page `j`. After getting a new pagerank vector, we swap it with the old one and repeat the computation. We stop when the vector converges, that is, for any index `i`, the difference between `oldp[i]` and `newp[i]` is bounded by a small constant `EPSILON`. The end result is the pageranks of all webpages we consider.

In more algorithmic terms, the computation can be divided into the following steps.

1. Initialize an array `float* oldp` of length `COUNT` with `1/COUNT` for each element.
2. Initialize an array `float* newp` of length `COUNT` with `(1-DAMPING)/COUNT`.
3. Add the contribution of hyperlinks to each element of the array `newp`, using the old pagerank array `oldp`.
4. Check if the differences of all pairs of `oldp[i]` and `newp[i]` are smaller than `EPSILON`. If so, terminate with `newp` containing the pageranks; otherwise, swap `oldp` and `newp` and go to Step 2.

In `pagerank.cu`, you have a sketch of a program that computes pageranks in a randomly generated network of webpages, on both CPU and GPU. When running, the program will print the 16 pages with largest pagerank and the timing. You can choose to implement first the GPU or the CPU side.

1. Write `__global__ void pr_init_gpu(float* pr)` for Step 1 and `__global__ void pr_damping_gpu(float* pr)` for Step 2.
2. Write `__global__ void pr_iter_gpu(const uint2* links, const float* oldp, float* newp)` for Step 3.
3. Write `__global__ void pr_conv_check_gpu(const float* old, const float* newp, uint32_t* conv)` for checking convergence. After the execution of this kernel, `*conv` should be 0 if and only if convergence is achieved.

4. Write `float pr_compute_gpu(const uint2* links, float* pr)` that controls the whole computation. It should return the number of **seconds** spent on GPU computation, and `*pr` should point to a host array with computed pageranks.

5. Complete the corresponding functions on the CPU side.

6. Compare the results. What do you observe? Try to explain with online resources.

7. **(Bonus, hard)** Produce a program without anomalies just observed.