GPU Programming in Computer Vision: Day 4

Date: Thursday, 11. September 2014

Please work in groups of 2–3 people. We will check your solutions tomorrow. Please be prepared to present your solution and explain the code. The general code requirements from exercise sheet 1 still apply. The bonus exercises are not mandatory.

Exercise 14: Denoising (8P)

Output: same number of channels as input image. Input: general number of channels.

Compute a minimizer u of the *denoising energy* with Huber-regularization,

$$\min_u \int_\Omega (u-f)^2 + \lambda\, h_\varepsilon\big(|\nabla u|\big)\, dx, \qquad \text{with} \quad h_\varepsilon(s) := \left\{ \begin{matrix} \frac{s^2}{2\varepsilon} & \text{if } s < \epsilon \\ s - \frac{\varepsilon}{2} & \text{else} \end{matrix} \right\},$$

by solving the corresponding Euler-Lagrange equation:

$$\frac{\partial E}{\partial u} = 2(u-f) - \lambda \operatorname{div}\Big(\widehat{g}\big(|\nabla u|\big)\nabla u\Big) = 0, \qquad \widehat{g}(s) := h'_\varepsilon(s)/s = \frac{1}{\max(\varepsilon, s)}.$$

1. Add Gaussian noise to your input image, with some noise variance $\sigma > 0$. For this, right after loading the input image, use the `addNoise` function from `aux.h`. You can set the noise level to e.g. $\sigma = 0.1$.
2. Compute the minimizer u by implementing the *Jacobi method* in several steps:
   (a) Compute $g = \widehat{g}(|\nabla u|)$. Reuse your code from exercise 11.
   (b) Compute the update step for u using the discretization from the lecture.
   (c) Compute N iterations and visualize the result. Test with different values λ. Check experimentally how many interations you need to achieve convergence (i.e. until visually there are no significant changes anymore).
3. Compute the minimizer u by implementing the *red-black SOR method* in several steps:
   (a) Compute g as above.
   (b) Compute the update step for u.

   The red and black updates are essentially the same, so write *only one* kernel, which will compute either the red-update or the black-update depending on a parameter. Note that you now also have a parameter $0 \leq \theta < 1$ for the SOR-extrapolation.

   Hint: The SOR update step is essentially the same as for Jacobi. The only change is that there is an additional θ-extrapolation, and that the update is performed only in certain pixels.
   (c) Compute N iterations and visualize the result. Test with different λ values, and also with different values of $0 \leq \theta < 1$.
   (d) Check how many interations you need for convergence. Do you observe a speed up when using SOR, compared to the Jacobi method?

Exercise 15: Parallel Reduction & Energy Computation (4P)

Output: Energy of the iterations. Input: general number of channels.

Write code to compute the sum of a float array u with n elements:

$$s = \sum_{i=1}^{n} u_i$$

Do so by performing a parallel reduction. In particular, complete the following tasks:

1. Implement a parallel reduction algorithm yourself. Calculate the sum of a `float` array with $n = 10^5$ elements to test it.
2. Use the library function `cublasSasum`. Compare the performance of your implementation with the performance of the library function.
3. Using either the library function or your own implementation, compute the denoising energy from the previous exercise after every iteration of the denoising algorithm.
4. Compare how quickly the Jacobi and SOR (for different θ) method drop the energy. Both methods should converge to the same energy.

Exercise 16 (Bonus): Histograms (requires CC $\geq$ 1.2) (3P)

Output: Histograms of the channels. Input: general number of channels.

First make sure that the computer you're using has a graphics card with CC ≥ 1.2, otherwise work remotely via `ssh`. Since we introduced a new function to the framework for this exercise, please clone the repository again (`git clone`) or update it (`git pull`).

Compute the intensity histogram with 256 bins of the input image using atomic operations. Make sure to compile your code with the flag `-arch=sm_12`. Proceed in the following way:

1. Represent the histogram as an integer array of length 256 and initialize it to zero. Write a kernel where each thread performs a global memory `atomicAdd` on the bin corresponding to the intensity of that pixel.
2. Visualize the histogram. You can use the `showHistogram256` function provided by the framework (`aux.h`).
3. Right now, the kernel is quite slow. Why? Think of a way to improve the performance of your kernel by using shared memory and shared memory atomics. Compare your solution to the naive version from 1.