GPU Programming in Computer Vision: Day 3

Date: Thu, 6 March 2014

Please work in groups of 2–3 people. We will check your solutions tomorrow after the lecture. 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 10: Linear Diffusion (3P)

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

Implement the Laplace diffusion

$$\partial_t u = \Delta u.$$

1. Use the diffusion discretization from the lecture. Instead of computing $\operatorname{div}^-(\nabla^+ u)$ in several steps, use the explicit discretization of Δ from the lecture.
2. Compute N iterations of the diffusion and visualize the end result. Experiment with different time steps τ and different numbers of iterations N.

   A necessary condition for convergence is $\tau < 0.25$. What do you observe for $\tau > 0.25$? What happens for a very large N?
3. Compare the result to Gaussian convolution $G_\sigma * u$ with $\sigma = \sqrt{2\tau N}$. What do you observe?

Exercise 11: Nonlinear Diffusion (7P)

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

Implement the Huber-diffusion

$$\partial_t u = \operatorname{div}\Big(\widehat{g}\big(|\nabla u|\big)\nabla u\Big),$$

with $\widehat{g}(s) := \frac{1}{\max(\varepsilon, s)}$. Here the gradient is computed from all channels of u:

$$|\nabla u(x,y)| \;=\; \sqrt{\sum_{c=1}^{n_c} |\nabla u_c(x,y)|^2} \;=\; \sqrt{\sum_{c=1}^{n_c} \big((\partial_x u_c)(x,y)^2 + (\partial_y u_c)(x,y)^2\big)}\;.$$

Implement this in several steps:

1. Use forward differences to compute the derivatives $v_1 := \partial_x^+ u$ and $v_2 := \partial_y^+ u$.
   Reuse your code from exercise 5.
2. Compute the diffusivity g from v_1, v_2. Reuse your code from exercise 5. Use a "`__host__ __device__`" function for $\widehat{g}$.
   Hint: Note that g is *scalar*, there is only one value g, which is shared for all channels.

3. Multiply v_1, v_2 by g, and store the result again in v_1, v_2.
   If you want, you can combine steps 2 and 3 into a single kernel. Note that then you don't need an array for g, because you can compute g locally in the kernel.

4. Use backward differences to compute the divergence: $d := \operatorname{div} \binom{v_1}{v_2} = \partial_x^- v_1 + \partial_y^- v_2$.
   Note that d has n_c channels, just as v_1, v_2 and u. The divergence operation is applied channelwise.

5. Compute the update step for u, update all of the n_c channels in a single kernel. You can implement this as a separate kernel, or as part of the div-kernel from step 3.

6. Compute N iterations of the diffusion and visualize the end result. Experiment with different time steps τ, number of steps N, and ε values.

   A necessary condition for convergence is $\tau < 0.25/\widehat{g}(0) = 0.25\,\varepsilon$. Start with $\varepsilon = 0.01$, and $\tau = 0.2\,\varepsilon$.

7. Try using a different diffusivity function:

   (a) $\widehat{g}(s) = 1$.
   (b) $\widehat{g}(s) = \exp(-s^2/\varepsilon)/\varepsilon$.

   How does the result change in each case?

Exercise 12 (Bonus): Mandelbrot fractal (5P)

Output: grayscale. Input: No input image.

You can use CUDA to generate a very well-known fractal: *the Mandelbrot set.*
For this, we need to work with *complex numbers*, which can be viewed as points in $\mathbb{R}^2$.

The Mandelbrot set M is a subset of $\mathbb{R}^2$: every complex number $c \in \mathbb{R}^2$ either lies in M (and is colored as black), or not (and is colored as white). One can construct successive approximations to M by running the following pseudo-code: Input is a complex number c which will depend on the location in the output image:

$n := 0$;
$z := c$; // z and c are complex numbers
`while` ($|z| < 2$ and $n < N$) {
    $z := z^2 + c$;
    $n{+}{+}$;
}

If the loop is exited *before* the maximum number N of iterations is reached, then $c \notin M$. Otherwise, c lies in a neighborhood of M. Increasing N yields better approximations.

Implement this iteration for a rectangular subset of $\mathbb{R}^2$:

1. Choose a rectangular subset $b \subset \mathbb{R}^2$ for which you want to compute the fractal. Specify it by choosing a center point and a radius. A good start is $(-0.5, 0) \pm 1.5$.

2. b is discretized into a number of pixels. Choose the desired output image dimensions, e.g. $w = 640$ and $h = 480$. Think of how you would map the pixels to the complex numbers in the region b. The mapping must be *orthogonal*, i.e. there must be *no skewing* in the y-direction.

3. Implement the above iteration for each pixel. Use the build-in type `float2` to represent complex numbers. Implement the multiplication, addition, and the absolute value operation as "`__host__ __device__`" functions.

4. Set the grayscale value in each output pixel to $1 - \frac{n}{N}$.

5. Experiment with different values of N. Try some other regions b to "zoom in" into the fractal, for example $(-0.773, 0.1175) \pm 0.005$.