GPU Programming in Computer Vision: Day 2

Date: Wed, 5 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.

Exercise 7: Convolution (continued) (11P)

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

In exercise 6 of sheet 1 you have implemented the convolution $G_\sigma * u$ using *global memory* for u, and global memory for the kernel $k := G_\sigma$.

1. Finish exercise 6.

Implement the convolution $G_\sigma * u$ on the GPU, now using:

2. *Shared memory* for the image u, but still global memory for k. To solve this task, use shared memory in the following way:
   (a)
   (b) To compute the convolution in an output pixel (x, y) one needs values of the input image in (x, y) and also in the neighboring pixels. For each kernel block, load into shared memory all input image values needed to compute the convolution in every pixel of this block. Note that overall there are *more* input image values to load than the size of the block, see Figure 1 on the next page. Set the size of the shared memory array accordingly.
   (c) When loading into shared memory, make the read accesses to global memory as coaleasced as possible (for instance, wherever possible, neighboring threads should access neighboring memory regions).
   (d) At image borders, use clamping.
   (e) Don't forget to use `__syncthreads()` after you've finished loading the data into shared memory.
   (f) The actual computation of the convolution may only use data from the shared memory (no global memory accesses allowed).
   (g) Use dynamic allocation of shared memory. Make sure you allocate exactly the right amount of shared memory for your kernel, and not more than needed.
   (h) For multi-channel images, apply the above procedure in a loop (within the kernel) separately for each channel. When you start processing each new channel, also synchronize *before* you begin writing to the shared memory.
3. *Texture memory* for the image u, but still global memory for k.

   Since we work with multi-channel images, but CUDA textures allow only one channel, define the texture as having width w and height $h \cdot n_c$. Here we use the fact that the channels are arranged in memory one after another. Therefore we can view a multi-channel image u as an n_c-times larger grayscale image u_1, defined by $u(x, y, c) = u_1(x, y + h \cdot c)$.

4. Pick some memory variant for the input image u (global, shared, or texture), and use *constant memory* for the kernel k. To define the CUDA constant kernel array, assume a maximal kernel radius $r_{\max} = 20$.

5. For some fixed $\sigma > 0$, compare the run times for the different memory versions. Which combination is the fastest? How much faster is it compared to the CPU version?

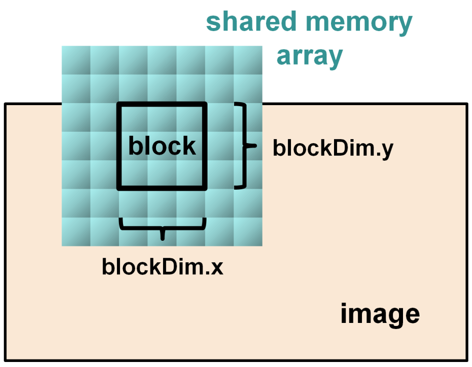

Figure 1: Shared memory array for convolution

Exercise 8 (Bonus): Structure Tensor (5P)

Output: three grayscale images. Input: general number of channels.

For an input image u, the smoothed version is defined as $S := G_\sigma * u$. The *structure tensor* T of u is defined at each pixel (x, y) as the smoothing

$$T := G_\sigma * M$$

of the matrix

$$M := \nabla S \cdot \nabla S^\top = \begin{pmatrix} (\partial_x S)^2 & (\partial_x S)(\partial_y S) \\ (\partial_x S)(\partial_y S) & (\partial_y S)^2 \end{pmatrix}.$$

The entries of M are scalar products over the n_c channels:

$$\big((\partial_x S)^2\big)(x, y) := \sum_{c=1}^{n_c} (\partial_x S_c)(x, y)^2, \quad \big((\partial_y S)^2\big)(x, y) := \sum_{c=1}^{n_c} (\partial_y S_c)(x, y)^2,$$

and

$$\big((\partial_x S)(\partial_y S)\big)(x, y) := \sum_{c=1}^{n_c} (\partial_x S_c)(x, y) \cdot (\partial_y S_c)(x, y).$$

Compute the structure tensor. Reuse your kernels for the convolution, don't write new kernels for this. You can use just the global memory for everything for convenience.

Implement this in several steps:

1. Compute $S = G_\sigma * u$.
2. Compute $v^1 := \partial_x S$ and $v^2 := \partial_y S$ using the *more rotationally symmetric* derivative discretizations $\partial_x^r, \partial_y^r$ as given in the lecture. Note that v^1, v^2 and S each have n_c channels.
3. Compute the matrix M at each pixel. The output should consist of three grayscale images m_{11}, m_{12}, m_{22}, corresponding to the three independent components of M at each pixel.
4. Compute $T = G_\sigma * M$ by convolving the three grayscale images m_{11}, m_{12}, m_{22}.
5. Visualize the grayscale images m_{11}, m_{12} and m_{22}. For this, you will need to define three new output images in the code framework.
   Hint: You will need to scale up these images, otherwise they will appear too dark because m_{11}, m_{12}, m_{22} are usually very small. To multiply an OpenCV `cv::Mat` image `m` by a scalar factor `f`, use `m *= f;`

Exercise 9 (Bonus): Harris Corner Detector (3P)

Output: color. Input: general number of channels.

Detect corners in the image using the structure tensor T from the previous exercise:

1. Compute the value
   $$H := \det(T) - \kappa\big(\operatorname{trace}(T)\big)^2$$
   in each pixel, with a fixed parameter $0 < \kappa < 0.25$. Implement this using a call to a `__device__` function with the three entries m_{11}, m_{12}, m_{22} of the structure tensor and the parameter κ as input values.
   Hint: The determinant is defined as $\det(T) = m_{11}m_{22} - (m_{12})^2$, and the trace as $\operatorname{trace}(T) = m_{11} + m_{22}$.
2. Pixels (x, y) with $H(x, y) > \theta$ are known as *Harris corners*, with a parameter $\theta > 0$. Produce and visualize the output
   $$u^{\text{out}}(x,y) := \begin{cases} (1,1,1) & \text{if } H(x,y) > \theta \\ 0.4 \cdot u(x,y) & \text{otherwise.} \end{cases}$$
3. The parameters $\sigma > 0$ for the smoothing, $0 < \kappa < 0.25$ for the detector H, and $\theta > 0$ for the corners, all depend on the input image u. Experiment with different values. As a start, choose $\sigma = 0.25$, $\kappa = 0.2$ and $\theta = 10^{-7}$, for the input image `bird.png`.