GPU Programming in Computer Vision: Day 1

Date: Monday, 8. September 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.

Download the Code Framework

In your home directory, execute:
`git clone https://svncvpr.in.tum.de/git/cuda_ss14`
The framework will be located in the folder `cuda_ss14`.

The framework shows how to use OpenCV to load/save/display images, access the camera, measure the run time, and process the command line parameters.
Compile: `make`
Run: `./main`

Copy the folder `framework` for each new exercise.
Reuse the kernels you have previously written as much as possible.

General Code Requirements for the Exercises

- Keep your code as general as possible. It must be applicable for images with an arbitrary number of channels n_c (if not stated otherwise).
- Always comment your code.
- Whenever new parameters are introduced, always write the corresponding `getParam` call, to be able to read in these parameters from command line arguments.
- Always include code for measuring run times and test how much time your overall computation for the exercise takes.
- When finished, test on several still images. If you want, also test on live webcam stream (uncomment `#define CAMERA`).
- Always use the macro `CUDA_CHECK` after each CUDA call, e.g.
  `cudaMalloc(...); CUDA_CHECK;`
- Hint: Multi-channel images are layered: access imgIn$(x, y, \text{channel } c)$ as
  `imgIn[x + (size_t)w*y + (size_t)w*h*c]`
- Always use a variable (of type `size_t`) for an index which you need *more than once*, e.g.
  `size_t ind = x + (size_t)w*y + (size_t)w*h*c;`
- Always cast to `size_t` in integer products when computing array indices or image sizes

Exercise 1: Check CUDA and the installed GPU (1P)

1. Open a terminal and check whether CUDA is installed: `nvcc --version`. Which version is installed?
2. Go to the "CUDA samples" folder[1] and run `deviceQuery`. Find out the following:
   (a) name of the installed GPU and its compute capability ("CUDA Capability")
   (b) number of multiprocessors and CUDA cores
   (c) amount of global memory
   (d) max. amount of registers and shared memory per block

Exercise 2: First CUDA Kernels (3P)

Implement the following CUDA kernels:

1. In `basic/squareArray.cu`, complete the CUDA code for squaring an array on the GPU. Implement the square operation as a `__device__` function. Compile with
   `nvcc -o squareArray squareArray.cu`
2. In `basic/addArrays.cu`, complete the CUDA code for adding two arrays on the GPU. Implement the addition operation as a `__device__` function.
3. Now, compile both files with (similarly for `addArrays`):
   `nvcc -o squareArray squareArray.cu --ptxas-options=-v`
   How many registers are used by your kernels?

Exercise 3: Gamma Correction (4P)

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

Perform gamma correction on the colors of the input image: $u_c^{\text{out}}(x,y) = u_c(x,y)^\gamma$, $\gamma > 0$ for each pixel $(x,y) \in \Omega$ and for each channel $c \in \{1,\dots,n_c\}$.

1. Write the CPU version. Keep your code general, so that it can process grayscale ($n_c = 1$) as well as color images ($n_c = 3$). Test on several input images, with and without the `-gray` parameter. Then test on live webcam images (uncomment `#define CAMERA`).
2. Write the GPU version. Test on still images and on the webcam stream.
3. Compare the CPU and GPU run times on still images. Average the run times over `repeats` ≥ 1 repetitions and experiment with different values of `repeats`. For the GPU version, first measure all operations, and then only the kernel executions excluding `alloc/free/memcpy`. What do you observe?
4. Experiment with several different block sizes for the kernel launch, starting with $(32, 8, 1)$. Make sure that the overall number of threads per block is a multiple of 32. For which block size is the run time minimal?

[1] `/work/sdks/cudacurrent/samples/C/1_Utilities/deviceQuery`

Exercise 4: Image Thresholding (2P)

Output: RGB image. Input: general number of channels.

Compute a thresholded version of the input image, defined for a fixed threshold $T \in [0, 1]$ and two colors $c_1, c_2 \in \mathbb{R}^3$ as follows:

$$u^{\text{out}}(x, y) = \begin{cases} c_1 & \text{if } \frac{1}{n_c} \sum_{c=1}^{n_c} u_c(x, y) \geq T \\ c_2 & \text{else.} \end{cases}$$

1. Write only the GPU version, and use a `__device__` function to get the result (the input of this function should be $\frac{1}{n_c} \sum_{c=1}^{n_c} u_c(x, y)$, and output $u^{\text{out}}(x, y)$). Keep your code general, so that it can be applied for general numbers of channels n_c.

2. Measure the kernel execution time averaging over several repetitions.

Exercise 5: Linear Operators (4P)

Output: grayscale. Input: general number of channels.

Write code for computing the gradient of an image and the divergence of a vector field. Then compute the absolute value of the Laplacian $\Delta u = \text{div}(\nabla u)$:

$$|\Delta u(x, y)| = \sqrt{\sum_{c=1}^{n_c} |\Delta u_c(x, y)|^2}$$

Write only a GPU version. As usual, write your code for a general n_c. Implement this in several steps:

1. Write a kernel which computes the gradient $v^1 := \partial_x^+ u$ and $v^2 := \partial_y^+ u$ given an input image u. The images v^1 and v^2 have the same number of channels as u, and ∂_x^+ and ∂_y^+ are applied channelwise.

2. Write another kernel which computes the divergence $w := \partial_x^- v_1 + \partial_y^- v_2$ of a given vector field v. The image w has the same number of channels as v_1 and v_2. The operators ∂_x^- and ∂_y^- are applied channelwise.

3. Write a third kernel which calculates at each pixel (x, y) the ℓ_2-norm across the color channels:
$$\sqrt{\sum_{c=1}^{n_c} u_c(x, y)^2}.$$

4. Finally use all three kernels to compute the absolute value of the Laplacian. Visualize the result.

Exercise 6: Convolution (6P)

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

Implement the convolution $G_\sigma * u$ of an input image u with a Gaussian kernel G_σ.
Use GPU global memory for everything.

1. Compute the kernel $k := G_\sigma$ on the CPU. Normalize so that the values sum up to 1. For a general variance $\sigma > 0$ set the kernel window radius to $r := \texttt{ceil}(3 \times \sigma)$ (i.e. round up).
2. Visualize the kernel using OpenCV. For visualization, define a copy k' which is equal to the kernel k but is scaled so that the maximum value is 1. Note that the kernel can be visualized as a grayscale image with width = height = $2r + 1$. *Remark:* For this, you will need to define a new OpenCV output image in the framework.
3. Compute the convolution $k * u$ on the CPU. The convolution is done channelwise on u. When the convolution requires values of u in pixels outside of the image domain, use clamping. Visualize the result.
4. Copy the kernel k computed in step 1 from the CPU to the GPU memory. Compute the convolution $k * u$ on the GPU. Use a single kernel execution to process all channels. Visualize the result.
5. Experiment with different values of σ on still images, compare the run times
6. Test on webcam images.

Exercise 7 (Bonus): cuBLAS (1P)

During the project phase it might be convenient to make use of additional libraries[2]. For example, the cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA runtime.
Use the `cublasSaxpy` function to add two float arrays on the GPU similarly as in Exercise 2. To obtain a cuBLAS handle use `cublasCreate` and for cleanup use `cublasDestroy`.

[2] `https://developer.nvidia.com/gpu-accelerated-libraries`