Lab assistant: Andreas Axelsson

Contemporary Computer Architecture
Lab 4 CUDA + Histogram

Introduction

The NVIDIA Jetson Nano is a small, powerful computer that enables the development of millions of new small, low-power AI systems. It's suitable for edge AI applications and research purposes. In this second lab we will start to explore the GPU of the Jetson Nano. Using CUDA (which is an acronym for Compute Unified Device Architecture) we can exploit the parallel power of the many cores found in the Nvidia GPU.

Learning Outcomes

By the end of this lab session:

- Learn how to implement the calculations of an intensity histogram of an image.
- Use shared memory to efficiently share part results between threads.
- Synchronize threads to maintain consistency and coherence between threads.
- See how parallel code can speed up calculations.
- Use profile tools to measure performance.

Setup

In this lab we will continue to use the Raspberry Pi camera to acquire images that we can do calculations on before we display them on the screen. To ensure we don't underexpose (get a dark noisy image) or overexpose (saturates as maximum intensity) the image, it can sometimes be very useful to calculate and display a histogram. Besides the help code posted in the lab-pm there is no additional setup or preparation required for this lab. But it will be very useful to read through the lab thoroughly in advance and look up things that you find unclear.

Exercise 1

In the first exercise you should familiarize yourself with histograms. Histograms are a plot of a discrete distribution function; thus, they show the relative occurrences of different values in a large set. In this case we are looking at the intensity values in an image. Each pixel can have an intensity value (grayscale) between 0 and 255. We can thus implement using a 256 long vector / array, and then for each pixel check its intensity value, and increase the corresponding value in the vector. As an example, if the current pixel in our loop has the intensity value of 137, we look at the histogram vector at index 137 and increase that value with one, and then write it back in the vector. Do this operation for each pixel in the image and you end up with an array with the number of occurrences for each intensity.

Let us investigate and answer some questions. Assume your image is a 1280 x 720 pixels RGB image, where each component is 8 bits. You have already created a kernel that can calculate the grayscale image as 8-bit size.

Questions:

1) Datatype

   What datatype is necessary for the histogram vector? ____________________

   hist_t histogram[256];

2) How many pixels does the image contain? ____________________

3) What is the maximum and minimum number of occurrences an intensity value can have in an image? ____

   (In other words, what is the maximum and minimum values we can find in a histogram)

4) Sometimes it can be valuable to calculate the histogram for each component (R, G and B) in the image. How can that be achieved? _______________________________

5) How do you prevent race condition when calculating the histogram? ____________________

Back to your implementation. You should reuse your grayscale image kernel from the previous lab. Using that kernel you now have a grayscale image in a new vector, after launching the grayscale kernel. In fact, you should reuse the last exercise from the last lab as the basis for this lab exercise 1. Make a folder for the new lab and copy and rename the *.cu file from the previous lab to ex1.cu.

Next step is to design your own calcHistogramKernel, which takes the image as the first argument (uchar4* type), and as second argument the pointer to the histogram vector (hist_t* type), and as third and fourth the width, and height of the image.

You may design your kernel either as a 1D kernel, or a 2D kernel, and you may also design it to be launched with enough blocks and threads to have each pixel handled by one thread, or you can have threads handling multiple pixels. Regardless of what path you take, make sure to try to structure it well, and for speed.

For exercise 1, you should not use the shared memory, but try to see how fast / slow it is to update the global memory directly for each time you update a bin in the histogram vector.

Write your kernel and try it out. Don't forget to zero the histogram vector before you calculate the histogram.

Bonus exercise: Is there a way to zero the vector inside the kernel?

____

If your implementation works, after launching the kernel, you should now have a vector containing the distribution of intensity values in the current image.

Questions:

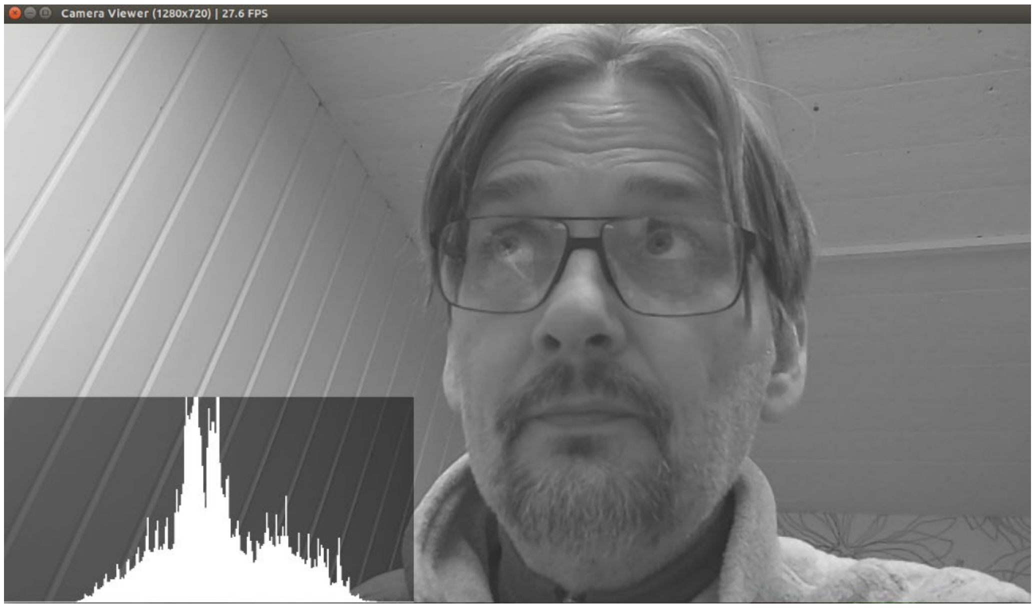
Without plotting the histogram, is there any way we can do some sanity calculations?
It is a 256 long array of numbers. What do these numbers represent?
Do some calculations to convince you and your neighboring lab partner that your code works!

What is the execution time of the kernel (use nvprof) ? ________________________________

Exercise 2

It is always nice to be able to visualize the histogram. Below is a kernel that modifies an image by altering the pixels. It plots the histogram as a graph in the lower left corner of the input image.

If you launch this kernel (use 256 threads, 1 block as it uses one thread to plot each bin of the histogram) you should get something like this image (except maybe not so handsome subject though! :-)



You can tell by the histogram that the lab assistant have a gray hair or two!

The plotting code:

```
__global__ void plotHistogramKernel(uchar4* image, int* histogram, int width, int
height, int max_freq)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    uchar4 white_pixel = make_uchar4(255, 255, 255, 255);
    uchar4 black_pixel = make_uchar4(0, 0, 0, 255);

    if (index < 256)
    {
        int freq = histogram[index] * 256 / max_freq;
        for (int i = 0; i < 256; i++)
        {
            int row = height - i - 1;

            if (i <= freq)
            {
                image[row * width + 2*index]   = white_pixel;
                image[row * width + 2*index+1] = white_pixel;
            }
            else
            {
                image[row * width + 2*index]   = black_pixel;
                image[row * width + 2*index+1] = black_pixel;
            }
        }
    }
}
```

The argument *max_freq* is a way to scale the histogram, as it corresponds to the full scale range of the graph. Around 20000 is a good starting point for *max_freq*. All the other arguments is as before.

As you can see, the kernel plots the graph in white on a completely black background. In the image above, you can see that the white graph is plotted on a semitransparent background.

TASK: How can this be accomplished? Modify your code accordingly!

Exercise 3

In the first exercise you calculated the histogram using global memory access. The task is to now try to speed up the histogram calculation task by using shared memory as much as possible.

One way is to have a local __shared__ histogram vector that the kernel updates in each SM, and then when finished, it adds these results to the total global histogram.

Here is the outline of the kernel code:

1) Define the local histogram vector __shared__ int histo_local[256];
2) Do CUDA code to clear this vector (make sure to use many threads in this to make it fast!)
3) Calculate the local histogram updating the histo_local
4) Update the global histogram

Just as in "Who Wants to Be a Millionaire" you are granted three helps; 50/50, phone-a-friend, and ask the audience.
In this case 50/50 is that you might discuss with the lab assistant. It is a 50% chance he helps you understand, and a 50% chance he confuses you even more!

Some pro-tips:

- Make sure you launch at least as many threads in a block as you have bins in your histogram. Then you can delegate per bin tasks more easily among the threads.
- Use __syncthreads() in a few strategic places

Did you succeed in speeding up the calculation of the histogram?

How fast was your new kernel according to nvprof? ______________________________

(The lab assistant got an average time of 776 microseconds. Can you beat that!)

Good work!!! Show the lab assistant your work and results. Lab OK: __________________