ci1009 Trabalho 2 1sem24

(version 1.3)

by W.M.N.Zola/UFPR

CUDA Image Blur using shared memory

Objective

The purpose of this lab is to implement an efficient image blurring algorithm for an input image using shared memory. The image in this lab/exercise will be transformed from RGB format to an “int per pixel” (ARGB) format before using your new ***blurKernelSHM***. After applying your new blur kernel, the final image needs to be converted back to the original RGB format, so that the image can be stored or compared to the result in the datasets.

Prerequisites

Before starting this lab, make sure that:

- You have completed lab1

Image Format

The input and output images are stored in the dataset directories in PPM P6 format. For this lab you will use the dataset (imageBlur). Newer (bigger) images can be added later to the datasets.

The output image FILE will not be generated by your program, you will use the output FILE provided in the datasets to compare the results you produce and verify the correctness of your kernels. The wbSolution used in this lab is in wb4.h file (which contains the code for correctness verification of your work.

Images loaded by the wb functions from PPM files produce RGB images as an array of unsigned char values, in sequence, 3 values per pixel

(with one red, green and blue unsigned char per channel, i.e. the 3 channels are interlaced) in the array. We will call this the ***rgb_input array*** in this description.

You need to provide CUDA kernels for transforming the ***rgb array*** to another array in a **“int per pixel” format**, where each pixel in the image is

aligned to a 32bit unsigned integer, with the same rgb values and the higher order byte filled with 0 (zero). This new array is called the ***argb array*** in this description. Each value in the argb array is an unsigned integer, corresponding to a pixel.

Given a pixel with components unsigned char components ***r***, ***g*** and ***b***, there is a corresponding integer with value ***v*** as:

```
unsigned int v = ((unsigned int)r << 16) + ((unsigned int)g << 8) + (unsigned int)b;
```

With the ***argb array*** format it will be simpler (and hopefully more efficient) to implement the ***blurKernelSHM*** in using shared memory.

Instructions

Edit the code to perform the following:

- allocate device memory
    - for rgb array and argb arrays (you may need more than one argb array)
- copy host memory to device (just the rgb array)
- initialize thread block and kernel grid dimensions
- invoke your 3 CUDA kernels that use shared memory:
  (***rgb2uintKernelSHM, blurKernelSHM, uint2rgbKernelSHM***) where:

rgb2uintKernelSHM

- will transform the rgb array to the equivalent array argb of unsigned interegers per pixel, uses shared memory

blurKernelSHM:

- will apply image blur algorithm to the “int per pixel” array and produce another “int per pixel array”, uses shared memory

uint2rgbKernelSHM:

- will transform the argb array back to the equivalent array rgb values, uses shared memory

- copy results from device to host (just the rgb array)
- compare results with the datasets
- deallocate device memory

.

Local Setup Instructions

You have to make a copy of each directory to your home account and work with your copy. The professor will give you instructions as to where/how to hand in your final solution (just the CUDA source file with the main code plus kernels, ready for compilation).

The executable generated as a result of compiling your lab solution must run using the following command:

./imageBlurSHM <input.ppm**> <**output.ppm**>**

<input.ppm> is the input dataset, and <output.pbm> is the output file with the results (also provided in the datasets). Some datasets are already generated in the lab directory.

Kernel types

Let's standardize the types of each kernel:

```
rgb2uintKernelSHM<<<GRID1, NT1>>>(unsigned int *argb, unsigned int *rgb);

blurKernelSHM<<<yourGrid, yourBlocks>>>(  unsigned int *argb_out,
                                          unsigned int *argb_in,
                                          int width, int height);

uint2rgbKernelSHM<<<GRID1, NT1>>>(unsigned int *argb, unsigned int *rgb);
```

where:

GRID1 is the number of blocks used by yor kernel. Suppose that MP is number of multiprocessors in the GPU. Use a unidimentional grid, and a grid with number of CUDA multiprocessors (SMs) in the GPU (details bellow). You can define GRID1 as a function of MP. For example, if your MP can run 2 threadblocks simultaneously:

#define GRID1 (MP*2)

NT1 is the number of threads per block you will be using. (use unidimentional blocks)

yourGrid, yourBlocks you decide if you use uni or bidimentional grid or blocks, just be warned the this has implications on how you are going to use the shared memory and the number of pixels each block can work at a time.

OBS: version 1.3

- rgb array type of the kernels to a (unsigned int *).
  wb4.h **still** reads the image as an array of unsigned chars,
- As it is now clear in the kernel prototypes, our kernels will read an **integer** at a time (from gpu global memory).

```
  Each thread may still use r, g, b pixel components as unsigned chars if you like,
  but these are register variables in the threads.
- You have to allocate the unsigned int rgb array in the device and use cuda to copy the packed
  bytes read by wb4.h to the device.


IMPORTANT:
- you can type cast the device rgb array to (unsigned char *) to use
  CUDA memcopy (to/from device).
- you have to allocate the unsigned int rgb array
  in the device as a multiple of sizeof(unsigned int) bytes, enough to fit all the bytes.
```

Use #define for each GPU type

Let's standardize by using defines for the grids and block of threads, we coud use variables or query the devices using CUDA, but these methods would make our work much harder. So let's agree on using defines. As an example, we have defines bellow for GTX680 and GTX480, with values for MP and good values for NT1 and GRID1 (where NT1 and GRID1 will be used in the **`rgb2uintKernelSHM`** and **`rgb2uintKernelSHM`** `kernels.`

`Please redefine the constants for YOUR current GPU as instructed in next pages.`

The number **MP** is only dependent on the CUDA compute capability. This is listed by the deviceQuery utility or in the cuda occupancy calculator (spreadsheet) available in our site (labs directory). The results of deviceQuery for some GPUs are also in the site. If you use another GPU, please send the professor re deviceQuery results for them and we will add the to the site.

The best value for **GRID1 and NT1** will be dependent on your kernels, basically the number of register per thread that your kernels are using or the amount of shared memory used by them. The compiler can inform you of the number of registers per thread when you compile with the nvcc option:

```
--ptxas-options=-v
```

With this number, and the amount of shared memory that the kernel uses, you can determine the best value for NT1 and GRID1. But, this will depend on GPU characteristics and the amount of shared memory your block of threads use. Further more, your allocation can also be parameterized and depend on number of threads per block also. You can use the CUDA occupancy calculator to obtain the best value for NT1 and GRID1 in our kernels (this will be further discussed in class).

// #DEFINES MOVED to the next page...

```
#define GTX480   480
#define GTX680   680
#define GPU GTX480
#if GPU == GTX480     // ←-- define YOUR current GPU here and its parameter constants BELLOW
    #define MP 15            // number of mutiprocessors (SMs) in GTX480
    #define GRID1    (MP*2)      // GRID size for rgb2uintKernelSHM and rgb2uintKernelSHM kernels
       #define NT1 768           // number of threads per block in the
                                 //   rgb2uintKernelSHM and rgb2uintKernelSHM kernels
                                 //    this is perhaps the best value for GTX480
#elif GPU == GTX680
    #define MP 8                    // number of mutiprocessors (SMs) in GTX680
    #define GRID1    (MP*2)         // GRID size for rgb2uintKernelSHM and rgb2uintKernelSHM kernels
    #define NT1 1024                // number of threads per block in the
                                    //   rgb2uintKernelSHM and rgb2uintKernelSHM kernels
                                    //    this is perhaps the best value for GTX680
#endif
```

by W.Zola/UFPR