TDT4200-PS5
CUDA Intro - Image Scaling

Jacob O. Tørring and Zawadi Svela

Deadline: Wed October 27, 2021 at 22:00 on BlackBoard
Evaluation: Pass/Fail

- All problem sets are to be handed in INDIVIDUALLY. You may discuss ideas with max. 1-2 collaborators and note them in the comments on the top of your code file, but code sharing is strongly discouraged. Preferably, seek help from TAs rather than get unclear/confusing advice from co-students!
- Code should not use other external dependencies aside from the ones we specified.
- You will need to get the correct output using your CUDA code in order to get a pass.

Introduction to CUDA – Image Scaling with Bilinear Interpolation

This exercise will serve as an introduction to CUDA. You will take a pre-written algorithm for image scaling that utilizes bilinear interpolation, and write a GPU version of this program.

Important: When you deliver your code, for this problem set and future ones, make sure that you only deliver the files explicitly asked for. This will only include code you have edited yourself, and not binary files, example input, etc. A consequence of this is that *you are not allowed to utilize external libraries not already included in the hand-out code.*

Program description

Bilinear interpolation [1] produces a value for an unknown function at a point by taking a weighted average of true values surrounding this point. In our case, we will use this to scale up an image. For each pixel in the new image, we map it to the corresponding point in the old image, and take a weighted average of the 1-4 corresponding pixels. A rough illustration of this mapping can be seen in Figure 1. The program utilizes the same pixel-structure used in the previous problem sets.

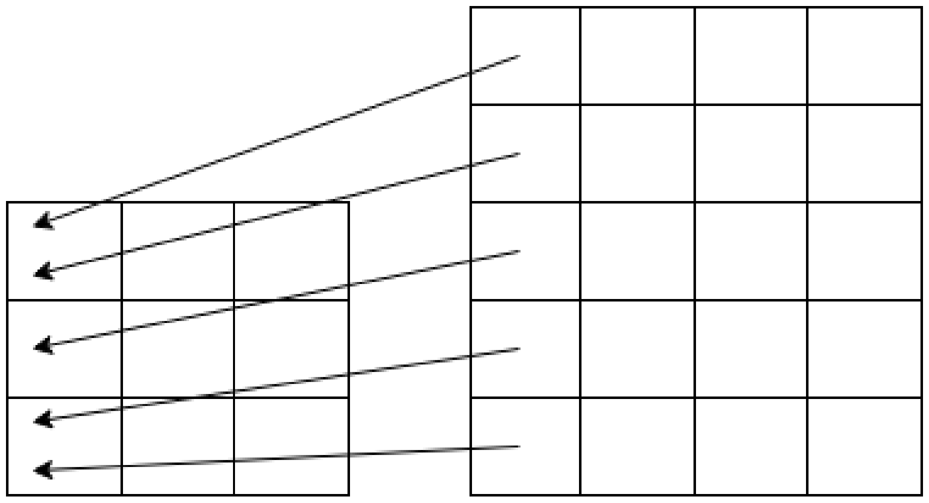

Figure 1: Illustration of mapping the first column of the upscaled image to corresponding positions in the original. Because the grids don't line up perfectly, we take a weighted average of the closest pixels.

The hand-out code can be run by compiling and providing any jpg/png image as input. You are only required to make it run correctly with the provided input image for the CUDA tasks, but it should produce the correct output for any integer scaling.

[1]Wikipedia article: `https://en.wikipedia.org/wiki/Bilinear_interpolation`

Setup

For this problem set we recommend using the Snotra cluster as described under Course Information if you do not have access to an Nvidia GPU.

Before you begin the tasks, you should start by testing the program. Compile the serial implementation using Make.

```
make main-serial

./main-serial input.jpg 2 5
```

This should create an image called "output.png", looking like the original but with the width doubled and the height quintupled.
Your CUDA implementation can be compiled and run using "main":

```
make main

./main input.jpg 2 5
```

Task

The following tasks are all marked with corresponding TODO comments in the handout code. After each task, you should ensure that the program compiles, though it might not run after some of the changes. For Part 1 and Part 3 it might be beneficial to store the size of the memory regions for the images that are allocated and transferred. You can then use these variables for all of the allocations and transfers.

Part 1

A Use cudaMalloc to allocate space for the input and output picture in device-side memory.

B Use cudaMemcpy to copy the data from the variables containing the host-side images to the device-side variables.

C Define the block size and grid size. The block size should usually be defined to be as large as possible to maximize performance. Define the block size to be the maximum size of 1024 (32*32). If this block size does not run on your GPU try a smaller block size, e.g. 16x16. The grid should be defined as a function of the block size and output image dimensions. There should then be a total number of threads in the grid matching or exceeding the total amount of pixels. Each pixel can then be mapped to an individual thread in Part 2.

Part 2

A Change the bilinear_kernel function to be a Kernel with launch parameters, using gridSize and blockSize. You need to change the code both where the function is defined and where the function is called in main (marked by TODO 3 a).

B Change the bilinear function to be a device-side function.

C Parallelize the double for loop in the bilinear_kernel function. Use the threadIdx.x, blockIdx.x and blockDim.x syntax to get the global index of the current thread in the x dimension(i.e. int x = threadIdx.x + blockIdx.x ∗ blockDim.x). Use the index in both x and y dimensions. Make sure to include a boundary check to check if the index is within the bounds of the image.

Part 3

A Copy the device-side data back to the host-side variables using cudaMemcpy.

B Free the malloced variables using free() and cudaFree().

Deliverables

Deliver the finished main_solution.cu file on Blackboard. It should run error-free and produce the correct output for any integer scaling. The problem set will be graded pass/fail.