hw2

ECE 3803

Jeff Epstein

Contents

1 Introduction

In this assignment, we will build on our CUDA skills. We will cover static allocation of device memory, signal-processing algorithms, and two-dimensional grids.

2 Assignment Part 1: Cypher with static allocation (20 points)

In the `cypher` program in the previous homework, we used `cudaMemcpy`, `cudaMalloc`, and `cudaFree` to manage *dynamically-allocated* device memory. An alternative approach is to use *statically-allocated* device memory. In this case, we will not use the above functions.

Your task is to revise your `cypher` program to use only static allocations on the GPU. Some points to remember:

- Your revised program must not use `cudaMemcpy`, `cudaMalloc`, and `cudaFree`. Instead, you should use the `__device__` attribute on global variables, and functions `cudaMemcpyToSymbol` and `cudaMemcpyFromSymbol`.
- The parameters `input_array` and `output_array` will now be global variables. They should be removed from `shift_cypher`'s parameter list. The new signature of that function will be `__global__ void shift_cypher(unsigned int shift_amount, unsigned int alphabet_max, unsigned int array_length)`.
- You will still need to use the host-based functions `malloc` and `free` for managing host memory, including data used by `host_shift_cypher`. Don't change `host_shift_cypher`.
- Ensure that your program runs correctly and that your revised GPU function produces identical results to the host version.
- Remember to use `CHECK_ERROR` where appropriate.

3 Assignment Part 2: Audio blur (40 points)

In this assignment, you will be implementing a convolution in CUDA. Specifically, we are going to take a signal (represented as a sequence of `floats`) and apply a Gaussian blur to it. Even more specifically, we will process audio data, on which the Gaussian blur will work as an audio filter creating a "blurring" of the sound. Because convolving with the Gaussian convolution kernel acts as an imperfect low-pass filter, the output file will have its higher frequencies attenuated.

(*Note*: we have a case of terminology overload, which could potentially cause confusion. The term "kernel" has two distinct meanings in this assignment: a "CUDA kernel" is an entity in programming, specifically a function with the `__global__` annotation. On the other hand, a "convolution kernel" is a term from signal processing, indicating a matrix to be applied to some signal. You will write a CUDA kernel that will apply a convolution kernel.)

A convolution is a way of combining two functions to produce a new function. In the below graphic, we represent one function as a blue line (the data) and another function as a red line (the convolution kernel). The convolution of these two functions yields a new function. To calculate the convolution, we're going to (in our imagination) slide one function along the other one, calculating the new value at each step.

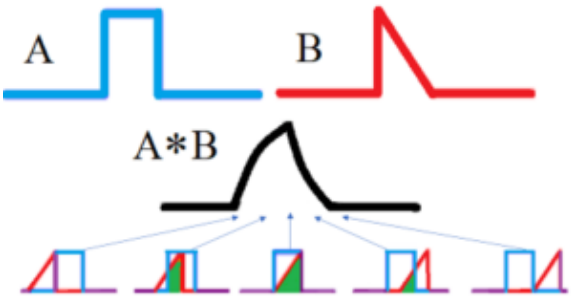

Figure 1: a convolution

In this exercise, we'll be convoluting two signals: one is an audio signal representing music or voice; and the other is a Gaussian filter, selected to apply a desired distortion. The resulting signal will be a distorted audio signal. By modifying the convolution kernel, we can modify the kind of distortion applied to the audio.

Sound is a waveform. To digitally record sound, we sample the waveform at discrete time intervals, each sample being a scalar value. An audio file is a stream of digital samples. In the diagram below, each vertical bar represents a sample approximating the waveform at a given point in time.

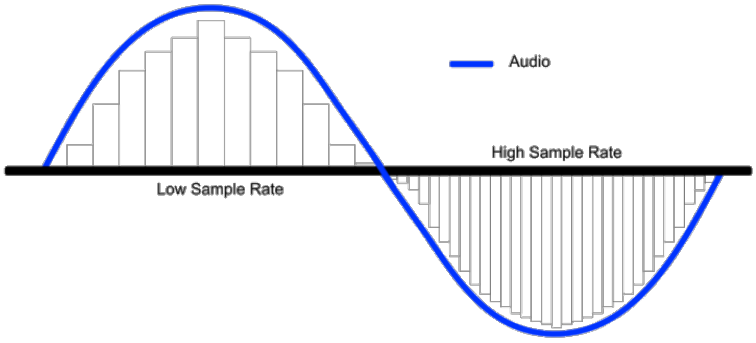

Figure 2: waveform with samples

Intuitively, Gaussian blur replaces each sample with a weighted average of its neighboring samples. The original sample's value receives the heaviest weight, and neighboring samples receive smaller weights as their distance to the original sample increases.

Examine the provided code for the `audioblur` program. You are given the following files:

- `audioblur.cpp` — main program code
- `audioblur.cu` — CUDA kernel code
- `audioblur.cuh` — CUDA header file
- `Makefile` — for building

The code for the Gaussian distribution can be found in the `gaussian` function in `audioblur.cpp`, corresponding to the following equation:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

which produces values given by the diagram below:

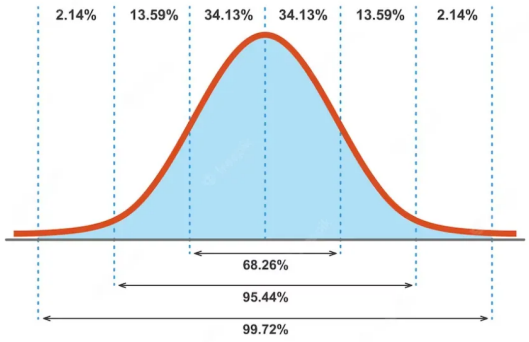

Figure 3: Gaussian distribution

You can verify these results by analyzing the value of the `blur_v` array.

When compiled, the program comes in two versions:

- `blur-audio` — load an audio (`.wav`) file, convolve its data with a Gaussian convolution kernel, and write the output to a new file
- `blur-noaudio` — generate random data, and convolve its data with a Guassian convolution kernel

Your task is to complete the GPU-based implementation of the algorithm by filling in the parts marked "TODO" in `blur.cu`. Do not modify any other code. Altogether, you will need to complete three functions:

- `cuda_call_blur_kernel` — host function, responsible for setting up memory, doing timing, and calling the CUDA kernel; you will be given the following parameters:
  - `blocks`, `threads_per_block` — the number of blocks and threads that your implementation should invoke the CUDA kernel on
  - `raw_data`, `n_frames` — an array of data storing audio samples, along with the number of audio samples
  - `blur_v`, `blur_v_size` — array storing the Gaussian filter, and the number of elements in it
  - `out_data` — an array into which your code should store the convolved data
- `cuda_blur_kernel` — the actual kernel to be run on the device, responsible for iterating through data indices

- `cuda_blur_kernel_convolution` — called by the kernel to handle one specific index, passed as `thread_index` from `cuda_blur_kernel`

Your implementation must take into account the `blocks` and `threads_per_block` parameters passed to `cuda_call_blur_kernel`.

You are provided with a CPU-based implementation of convolution in `audioblur.cpp`. You should refer to this implementation as you implement your GPU-based version. In essence, you are "translating" the sequential CPU-based version into a parallel version, as we have done in the example programs above.

The provided code will automatically test your implementation by comparing its results against the CPU-based implementation in `blur.cpp`. If your CUDA implementation of the convolution algorithm fails to produce the same results as the CPU-based version, the program will output a list of mismatched data points. Furthermore, GPU errors (such as memory-access errors) will be reported.

Your solution should work with any number of blocks and threads-per-block. To make sure it works, change the values passed as command-line parameters to the executables in the `Makefile` stanzas for `test-audio` and `test-noaudio`.

Use the provided `HANDLE_ERROR` macro (as in the example code) to check for the success of CUDA function calls.

You should be able to get a reasonable speedup (at least 10x with audio) relative to the CPU implementation.

You can use the provided `Makefile` to build and run the program. Use the command `make` to compile the executables. The `Makefile` also contains tests, which you can run with the command `make test`.

You are provided with a sample audio file `resources/example_test.wav`, containing an excerpt from J.S. Bach's Concerto for Violin & Oboe in C Minor, BWV 1060R, performed by the Orchestre du Festival de Prades. The automated tests will apply your CUDA kernel to this file and store the convoluted data in `resources/example_test_out.wav`. Comparing the "before" and "after" version of the music, you can hear the difference: the original is clear and vibrant, while convoluted version is "fuzzy" and distant, as if it were being played in another room. You may need to use headphones to appreciate the difference.

Note: the assignment uses the `libsndfile` library, which has been installed on the ICE cluster. If you are running and compiling this program on your own computer, you should install the appropriate library. On Debian and Ubuntu Linux systems, this can be accomplished with the command `sudo apt-get install libsndfile1-dev`.

4 Assignment Part 3: Image blur (50 points)

Now let's apply a similar strategy to a different type: instead of blurring audio, we will blur an image.

An image is a sequence of pixels, represented as an array of pixel color values. For this assignment, we will consider only square images of dimension `IMAGE_DIM`, so each array will have `IMAGE_DIM` $\times$

IMAGE_DIM elements in total.

Each element of the array represents an individual pixel's color. Pixels are stored in an array in row-major order. Each pixel's color is expressed by three scalar color channels: red, green, and blue. Each color channel is an `unsigned char`, having a value in the range $[0 \ldots 255]$. The displayed color of each pixel is determined by the intensity of each channel. For example, a black pixel would have the channel values $(0, 0, 0)$; a white pixel would have the values $(255, 255, 255)$; a red pixel would have the value $(255, 0, 0)$; bright yellow is $(255, 255, 0)$, and $(67, 93, 122)$ would be a grayish-blue. To store each pixel's three channels, we will use CUDA's `uchar4` type, which is defined as follows:

```
struct uchar4 {
    unsigned char x, y, z, w;
};
```

For our purposes, the `x` field will store the red channel; `y` the green channel; `z` the blue channel; and `w` will be ignored. (Why not use `uchar3`, since only three of the four fields matter? Because CUDA supports texture elements consisting only of 1, 2, or 4 scalars.)

For blurring each pixel, we will be using a simpler method than a Gaussian filter: we will just calculate the average pixel value from the surrounding pixels, within a square of size `SAMPLE_DIM`. For example, in the following diagram, if we want to calculate the new value of the red pixel, we will sum the three color channels of yellow pixels as well as the red pixel, and divide each channel by the number of pixels in the area, which in the case of this diagram is 25. For averaging pixel values, each channel is to be averaged separately.

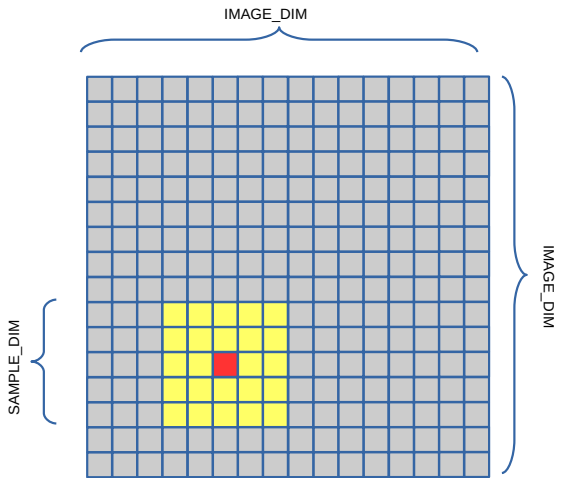

Figure 4: average of surrounding pixels

The kernel will be launched with a two-dimensional grid and two-dimensional blocks. Your kernel code will need to determine the correct array element to access, based on the current value of `threadIdx.x`, `threadIdx.y`, and the other special CUDA variables. For pixels near the edge (top, bottom, left, or right) of the image, the area of the kernel should "wrap around" to the other side, as shown below:

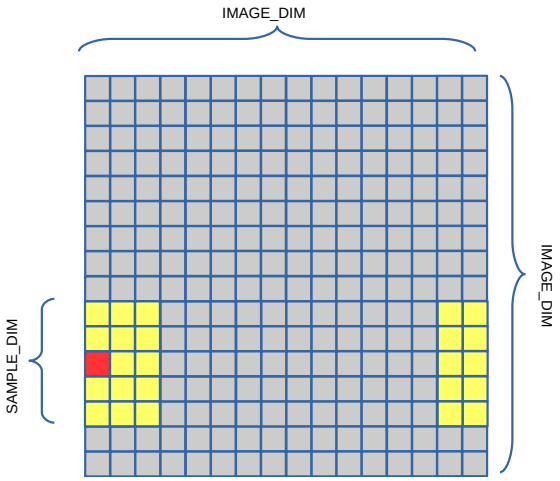

Figure 5: average of surrounding pixels near the edge

The program reads and writes image files in the `ppm` file format, which should be displayable by most image viewing software. If your computer doesn't come with software capable of displaying `ppm` files, you may need to download one. GIMP is one such option.

Your task is to complete the GPU kernels for performing the blurring. You will implement in total three kernel functions:

- `image_blur` — apply the blur to an image stored in device memory as an array
- `image_blur_texture1D` — apply the blur to an image stored in texture memory as a one-dimensional texture
- `image_blur_texture2D` — apply the blur to an image stored in texture memory as a two-dimensional texture

The code for the three kernels will be substantially identical. The only difference between them is the code for reading the input data: the first version will simply index into an array normally, while the other two versions will use CUDA's texture-access primitives. The code for creating the textures has been provided for you.

Compile the program with `make`. Run automated tests with `make test`. Your are provided with a sample image in `input.ppm`; the tests will create output images for each mode. View the images to ensure they are as expected.

5 Assignment Part 4: Questions (15 points)

Answer the following questions in a plain text file named `hw2.txt`.

1. What speedup was reported by the automated tests when you run `make test` on your audio blur program?

2. In the audio blur program, the kernel has both input and output arrays (`gpu_raw_data` for input, `gpu_out_data` for output). Why do we need both of these? In other words, why can't we do the convolution "in place" and simply mutate the input array?

3. What is the significance of the `GAUSSIAN_SIDE_WIDTH` macro defined in the `audioblur.cuh` header file? Specifically, what role does it play in the algorithm? If this number were increased to 100, what *audible* effect would you expect that to have on the program's output, and why? What *performance* effect would you expect that change to have, and why? Test your hypothesis and give the results.

4. Let's see if CUDA's type hints have a real impact on the compiled code. We know that if the compiler surmises that data is read-only, it will use a different method of accessing the data to ensure better cache use. We should be able to look at the generated assembly-language code and see which instruction is used.

   Compile your `imageblur.cu` into PTX assembly language using this command:

   ```
   nvcc --ptx imageblur.cu
   ```

   This will produce the file `imageblur.ptx` containing PTX assembly language. Open the file and look around. You should be able to make educated guesses about many of the instructions. For example, the `tex.1d.*` and `tex.2d.*` series of instructions are used to read one-dimensional and two-dimensional textures, respectively. Many instructions bear suffixes to indicate the type of their operands: for example `mov.u32` moves an unsigned 32-bit values into a register, while `tex.1d.v4.u32.s32` reads an unsigned 32-bit value from a one-dimensional texture as a four-element vector where the coordinates are specified as signed 32-bit values.

   Within the file, find the code for the function `image_blur`. Note that the name will be mangled according to C++ convention. If you have trouble finding it, look for the function whose parameter types match. Now find the memory load instruction (`ld.*`) that corresponds to the array indexing in your function. Give the full name of the instruction.

   Now let's see what happens if we try to convince the compiler that the `image` parameter is read-only. Mark that parameter using the `__restrict__` attribute, as we discussed, then recompile the file to PTX. Did the load instruction change? Give the new full name of the instruction.

   What about the `__ldg` intrinsic function? Remove the `__restrict__` attribute and do the memory load using CUDA's `__ldg` function, then recompile to PTX. Did the load instruction change? Give the new full name of the instruction.

   Does the use of the `const` modifier on the `image` parameter affect the results?

5. In a previous part of this assignment, we built a CUDA kernel to blur an image. Many common image manipulation operations can be described in terms of manipulation of the underlying pixel data. Write a CUDA kernel to be used in place of your `image_blur` kernel that will convert the given image to a grayscale representation; i.e. the image contains only shades of gray. Test your kernel and verify that it works on the test image. It should have the following form:

   ```
   __global__ void image_grayscale(uchar4 *image, uchar4 *image_output) {
       ...
   ```

```
}
```

Hint: your grayscale kernel needs to consider only one pixel at a time.

6 Rules

6.1 Development environment

Your code will be tested in our ICE environment, as described in the syllabus. Please test your code in that environment before submission. Your code must work with the operating system and CUDA development tool chain installed there.

You may use the CUDA libraries, as described in the ICE tutorial; as well as standard libraries that form part of the compiler. Do not use any other libraries.

6.2 Evaluation

Your submitted work will be evaluated on the following metrics, among others:

- your submission's fulfillment of the goals and requirements of the assignment. That is, does your program work?
- your submission's correct application of the techniques and principles studied in this course. That is, does your program demonstrate your understanding of the course material?
- your submission's performance characteristics. That is, does your program run fast enough?

Failure to satisfy the assignment's requirements, to apply relevant techniques, or to meet performance expectations may result in a grade penalty.

Only work submitted before the due date will be evaluated. Exceptions to this policy for unusual circumstances are described in the syllabus.

Ensure that your program compiles and runs without errors or warnings in the course's development environment. If your code cannot be compiled and run, it will not be graded.

In addition, you are obligated to adhere to the stylistic conventions of quality code:

- Indentation and spacing should be both consistent and appropriate, in order to enhance readability.
- Names of variables, parameters, types, fields, and functions should be descriptive. Local variables may have short names if their use is clear from context.
- Where appropriate, code should be commented to describe its purpose and method of operation. If your undocumented code is so complex or obscure that the grader can't understand it, a grade penalty may be applied.
- Your code should be structured to avoid needless redundancy and to enhance maintainability.

In short, your submitted code should reflect professional quality. Your code's quality is taken into account in assigning your grade.

6.3 Academic integrity

You should write this assignment entirely on your own. You are specifically prohibited from submitting code written or inspired by someone else, including code written by other students. Code may not be developed collaboratively. Do not use any artificial intelligence resource at any point in your completion of this assignment. Please read the course syllabus for detailed rules and examples about academic integrity.

7 Submission

Submit only your completed `cypher.cu`, `audioblur.cu`, `imageblur.cu`, and `hw2.txt` files on Gradescope. Do not submit binary executable files. Do not submit data files.