```
================================================================================
Question 1: Common Errors (20 points)
================================================================================
This class will make heavy use of low-level C constructs and concepts,
especially pointers and memory management.

As a "warm-up", here are a few quick samples of code and their intended
specifications. Each such piece of code is incorrect. Identify what is
wrong with the code, and how it should be fixed.

(Many of these problems allude to common errors encountered while writing
both GPU and CPU code.)

--------------------------------------------------------------------------------
1.1
--------------------------------------------------------------------------------
Creates an integer pointer, sets the value to which it points to 3, adds 2
to this value, and prints said value.

void test1() {
    //int *a = 3;
    int *a;
    *a = 3;
    *a = *a + 2;
    printf("%d\n", *a);
}

--------------------------------------------------------------------------------
1.2
--------------------------------------------------------------------------------
Creates two integer pointers and sets the values to which they point to 2
and 3, respectively.

void test2() {
    //int *a, b;
    Int *a, *b;
    a = (int *) malloc(sizeof (int));
    b = (int *) malloc(sizeof (int));

    if (!(a && b)) {
        printf("Out of memory\n");
        exit(-1);
    }
    *a = 2;
    *b = 3;
}
```

1.3

Allocates an array of 1000 integers, and for i = 0, ..., 999, sets the i-th element to i.

```
void test3() {
    //int i, *a = (int *) malloc(1000);
    int i, *a = (int *) malloc(sizeof(int)*1000);
    if (!a) {
        printf("Out of memory\n");
        exit(-1);
    }
    for (i = 0; i < 1000; i++) {
        *(i + a) = i;
       printf("%d\n",*(i + a))
    }
}
```

1.4

Creates a two-dimensional array of size 3x100, and sets element (1,1) (counting from 0) to 5.

```
void test4() {
    int **a = (int **) malloc(3 * sizeof (int *));
      for (int i = 0; i < 3;i++) {
            *(a+i) = (int *)malloc(100 * sizeof(int));
        }
    a[1][1] = 5;
}
```

1.5

Sets the value pointed to by a to an input, checks if the value pointed to by a is 0, and prints a message if it is.

```
void test5() {
    int *a = (int *) malloc(sizeof (int));
    //scanf("%d", a);
    cin >> *a;
    if (!*a)
        printf("Value is 0\n");
}
```

```
================================================================================
Question 2: Parallelization (30 points)
================================================================================

--------------------------------------------------------------------------------
2.1
--------------------------------------------------------------------------------
Given an input signal x[n], suppose we have two output signals y_1[n] and
y_2[n], given by the difference equations:
                y_1[n] = x[n - 1] + x[n] + x[n + 1]
                y_2[n] = y_2[n - 2] + y_2[n - 1] + x[n]

Which calculation do you expect will have an easier and faster
implementation on
the GPU, and why?
```

The first output calculation is faster since it does not depend on other output elements, but solely on input elements. Thus, the GPU can concurrently compute all output elements of y_1 without any of the elements having the need to wait for other y_1 elements to finish calculating.

```
--------------------------------------------------------------------------------
2.2
--------------------------------------------------------------------------------
In class, we discussed how the exponential moving average (EMA), in
comparison
to the simple moving average (SMA), is much less suited for parallelization
on
the GPU.

Recall that the EMA is given by:
        y[n] = c * x[n] + (1 - c) * y[n - 1]

Suppose that c is close to 1, and we only require an approximation to y[n].
How can we get this approximation in a way that is parallelizable? (Explain
in words, optionally along with pseudocode or equations.)

Hint: If c is close to 1, then 1 - c is close to 0. If you expand the
recurrence relation a bit, what happens to the contribution (to y[n]) of
the terms y[n - k] as k increases?
```

Y[n] can be approximated by solely x[n], since 1-c is close to 0 which makes the subsequent y[n-k] terms contribute little to y[n], as k increases.

```
================================================================================
Question 3: Small-Kernel Convolution (50 points)
================================================================================
```

Notes: The CPU time is actually faster than the GPU time, possibly because memcpy operations are included during time-taking, or maybe the Intel i7-8700K is that fast. Trying to increase the data length did not affect the ratio.

```
typedef unsigned int uint; Need this line else uint is undefined

while (thread_index < n_frames) { prevents going over the max number of data points.

// TODO: Update the thread index
thread_index += blockDim.x*gridDim.x;
```

Make thread work on the next datapoint. Meaning a thread does not stop working until it goes over the max datapoint limit of the while-loop above.

```
for (int i = 0; i < blur_v_size; i++) {
	if (i > thread_index)
		break;
	gpu_out_data[thread_index] += gpu_blur_v[i] * gpu_raw_data[thread_index - i];
	}
```

The if-statement prevents reading negative indices in the data-vector, x[-1] etc. Remember convolution is moving a sliding window that has been inverted across the data vector, summing the products of corresponding elements between the data-vector and inverted filter and writing that SoP into the output vector.

Eg. y[0] = x[0]*h[k]; y[1] = x[1]*h[k-1] + x[0]*h[k] …

Also, the filter is a Gaussian blur, since a low-pass filter in frequency domain is a brick-wall function. Hence in time-domain it is a Sinc-pulse, which can be approximated by a Gaussian.

The rest are standard practises like malloc, memcpy, run the kernel, then free.

Installing the sndfile library allows WAV file experimentation. The high-pitched violin sounds are attenuated after filtering.