# ECE 1570
# High Performance Computing

Take-home exam
Due date: Saturday, 17 December, high noon.

## Rules of the Game[1]

You may not discuss this exam with anyone except the guy who wrote it. No one. Not your parents. Not your significant other. Not your cat. Not even your pet goldfish. If you have questions, send email or zoom.

## The questions

1. Brainy format

   The Google Brain format (bfloat16) has 1 sign bit, 8 bits of exponent and 7 bits of mantissa. The exponent is represented as "bias 127" meaning that exponent 0 is represented by 127 (0x7F). So, that means the minimum exponent is -126 and the maximum exponent is ... 127.

   Just like the IEEE 754, when the exponent is maximum (0xFF) and minimum (0x00), there are special meanings:

   | Exponent | Mantissa | Meaning |
   |---|---|---|
   | 00 | zero | zero |
   | 00 | non-zero | Denormalized number |
   | FF | zero | $\infty$ |
   | FF | non-zero | NaN |

   (a) Give an example of a non-transcental number that can *not* be represented in bfloat16. Show why not.

   (b) Write a C program that computes sin (and cos). What is your angular resolution? Hint: The type declaration `short int` will give you a 16 bit integer.

[1]with apologies to Jean Renoir, director of "La Règle du Jeu"

2. Consider a complex array with the C type definition: `typedef struct complex { float real, imag; } complex`. Suppose you have a shared memory multiprocessor system where each processor has a cache. Examine the following code:

```
#define N 1024
// shared memory
static complex FFT_input[N];

float cabs(complex x) {
    return(sqrt(x.real*x.real + x.imag*x.imag));
}
// On Processor A
    for (int i=0, sum = 0; i<N; i++)
        sum += cabs(FFT_input[i]);
// On Processor B
    for (int i=0, sum = 0; i<N; i++)
        if (cabs(FFT_input[i]) > max) max = FFT_input[i];
// On Processor C
    for (int i=0, sum = 0; i<N; i++)
        if (FFT_input[i].real < 0) FFT_input[i].real = 0;
```

(a) Discuss the cache interactions of this code.

(b) What changes could you make to this code to improve the cache performance?

3. The *Floyd-Steinberg* algorithm is used for half-toning an image. From the “Source of All Knowledge” (Wikipedia), the algorithm is as follows:

```
for each y from top to bottom do
    for each x from left to right do
        oldpixel := pixels[x][y]
        newpixel := find_closest_palette_color(oldpixel)
        pixels[x][y] := newpixel
        quant_error := oldpixel - newpixel
        pixels[x + 1][y    ] := pixels[x + 1][y    ] + quant_error × 7 / 16
        pixels[x - 1][y + 1] := pixels[x - 1][y + 1] + quant_error × 3 / 16
        pixels[x    ][y + 1] := pixels[x    ][y + 1] + quant_error × 5 / 16
        pixels[x + 1][y + 1] := pixels[x + 1][y + 1] + quant_error × 1 / 16
```

where `find_closest_palette_color(oldpixel) = round(oldpixel / 255)`.

(a) Implement using OpenMP or CUDA.

4. Artificial Reverb

   Multitrack recording (invented by guitarist Les Paul and vocalist Mary Ford) overlays anechoic (e.g. rooms without any reflection) recordings. But the final result sounds artificial. To improve the sound, artificial reverberation is added. Inititally done in analog form, it can be done digitally. Remember convolution? Let $h(t)$ be the impulse response of the room and $x(t)$ be the anechoic recording. Then the new $y(t) = x(t) \odot h(t)$, where $\odot$ denotes time-domain convolution. But, this is quite a long sequence for a real recording and is $O(n^2)$. Let's examine the frequency domain. Then $Y(\omega) = X(\omega) * H(\omega)$. One additional complication is that this is *circular convolution* instead of linear convolution. But we'll address that.

   The proposed solution is to divide $h(t)$ into "reasonable" chunks (frames) of 1024 samples. An additional 1024 zero samples are appended to the input chunk. Then, using a 2048 point FFT, find $H_i(\omega)$, where $i$ is the chunk number. The $H(\omega)$ is a *constant*.

   So, then, the input $x(t)$ is cut (windowed) into same size frames (1024 samples), zero padded with 1024 zeroes and transformed with a 2048 point FFT. The resulting complex multiplication is done with $H_i(\omega)$, frame by frame. This will now look like the following diagram, assuming that $H(\omega)$ is 3 frames long:

| Frame 0 | $X_0 * H_0$ | | |
|---|---|---|---|
| Frame 1 | $X_1 * H_0$ | $X_0 * H_1$ | |
| Frame 2 | $X_2 * H_0$ | $X_1 * H_1$ | $X_0 * H_2$ |
| Frame 3 | $X_3 * H_0$ | $X_2 * H_1$ | $X_1 * H_2$ |

   Each time frame (row) adds up the component multiplies. And then uses the Inverse FFT to get the time domain signal. One additional complication: You need to remove the last 1024 samples to get a linear convolution. This leaves you with the 1024 output samples (this is "Overlap-Add").

   Implement this algorithm in a parallel fashion. Use any language / system you are comfortable using. You are provided two room impulses: one reasonable one (Office.ascii) and one very echoy one (Chapel.ascii). Feel free to truncate as needed. The Chapel response is quite long and shouldn't be tried unless you're *very* serious.

Here is how to proceed step by step from serial to parallel:

(a) Serial: One frame of $H$ and $X$

(b) Serial: Multiple frames

(c) Parallel: Multiple frames (most important achivement)

(d) Parallel: Implement Overlap-Add output (additional tidbit)

About testing: Remember that $x(t) \odot \delta(t) = x(t)$. So, what is the Fourier Transform of $\delta(t)$?