ECE 459 Winter 2022 Final Assessment

Instructions

General

1. This is a take-home exam and should be treated as such. You are expected to do the exam independently and may not collaborate with other people (classmates or otherwise).
2. This is an open-book exam, so you can consult your notes, the lecture materials, Google, Stack Overflow, man pages, etc.
3. If you take code from a source on the internet, make sure you cite it with a comment including the URL where you found it.
4. We will not answer questions on Piazza or via e-mail; if you need to state an assumption, do so.
5. Submit your PDF with the written questions in Crowdmark, and add and commit your code in gitlab.
6. Organize your files in your gitlab repository so the repository root has the README file and each question is in a subfolder, like the starter code (i.e., if question 3 is a programming question, the `q3/` folder is at the root level of your repository and contains question 3).
7. Be sure to submit your files on time.

Written Questions (Q1, Q2)

1. You can create your PDF using whatever software you like.
2. Answer the questions in the order of the exam, but also make it clear to the reader what question is being answered.
3. Use the amount of marks associated with a question as your guide for how much you should write; we recommend keeping it brief.

Programming Questions (Q3 - Q5)

1. You are allowed to modify the code, packages, and build files as you need, except you cannot change the output formats (for marking consistency).
2. Your code needs to run on `eceubuntu` (or `ecetesla` for CUDA) machines. Code that does not compile will result in a zero for that question.
3. You can use the compiler, code analysis tools, debuggers, etc.
4. If you are having technical issues with the ECE Servers, please e-mail `praetzel@uwaterloo.ca` (course staff are not admins on the machines).
5. As with the assignments, your local machine setup cannot be supported.
6. Please try to distribute your work across servers; `ecetesla` machines are the only ones that can run CUDA problems, so try using `eceubuntu` when the GPU is not needed.
7. Complete the `README.txt` to inform the marker about the server you used for each code question.
8. Please respect the directions to use a maximum of 4 threads; this helps reduce the server load to allow multiple people to work at once.
9. Server resources are limited and extensions will not be granted due to demand issues, so don't procrastinate.

1 Short Answer [30 marks]

Answer these questions using at most three sentences. Each question is worth 3 points.

(a) (Benchmarking). We often used `--warmup` for hyperfine to discard the first N runs, which are slower than others. Give a reason that the first run is likely to be unrepresentative of actual performance.

(b) (Queueing theory). Assume $\lambda = 1/150s$ and $s = 90s$ for a queueing theory model of a toaster (i.e. for toasting sliced bread) in a breakfast restaurant during the morning rush. In particular, the setup is that there is a cashier taking orders at some rate; assume that the rate is the maximal rate a human cashier can take orders from customers, and that every order includes a request for 1 piece of toast. Compute the resulting completion time average T_q. How can the restaurant improve T_q? Calculate the impact of that change.

(c) (Reduced-resource computation). One way that we could "do less work" is by setting a maximum time limit on a decision and proceeding with a default action if no decision is reached before that time. Give an example of a problem domain where this strategy is reasonable, the default action, and justify your selections.

(d) (Queueing theory). We often model arrivals of requests to a service using the Poisson distribution because we have a very large number of sources (users) each of which has a very small probability of initiating a request at any given instant. Does this distribution still make sense for "unusual" situations, like visits to Amazon.ca on Black Friday? Justify your answer.

(e) (Compiler optimizations). Another compiler optimization that we did not cover, but that compilers can do, is called *dead store elimination*. In this optimization, if the compiler determines that a write to an area of memory that is not read later (ever) can simply be skipped because it serves no purpose. Is this beneficial? Justify your answer.

(f) (Overhead). You are trying to measure the performance of a program. But this program is printing megabytes of output to the terminal. (i) Give two possible reasons that the timing information you're collecting may be inaccurate. (ii) To eliminate overhead, you experiment with redirecting output to /dev/null, but you find that the program is still slower than when you outright disable the output code. Why is that?

(g) (Inter-thread communication). While it is always possible to use either message-passing or shared memory, sometimes one is more appropriate than the other. Give an example of a concrete situation where it makes more sense to use shared memory, and say why shared memory is better.

(h) (Bottlenecks). If you're not careful, using a general-purpose memory allocator can be a bottleneck in a multithreaded context. (i) What's one possible reason for the bottleneck? (ii) What's one way to possibly avoid this bottleneck? (And don't forget to benchmark the performance before and after!)

(i) (AWS doing things for you). If AWS launches a new service that does something you've implemented, how do you know when/if it's time to switch? Name three considerations that you would take into account when deciding.

(j) (Rust/ECE/UW.) Make me a funny (safe-for-work) meme related to the Rust programming language, ECE 459, the Electrical & Computer Engineering Department, Faculty of Engineering, or the University of Waterloo generally. Put the picture in the PDF.

2 Self-Optimizing Payment Processing [15 marks]

In this question, it is the future and you are a senior software developer. Your software-as-a-service company uses a third party service to charge your customers. A simplified version of the workflow to pay the invoice is below:

```
fn process_invoices() {
    loop {
        let current_batch = get_next_invoice_batch();
        if current_batch.is_empty() {
            println!("Finished processing invoices.");
            break;
        }
        for mut invoice in current_batch.into_iter() {
            println!("Paying invoice {}.", invoice.id);
            let result = try_to_pay(&invoice);
            if result.unwrap().success {
                println!("Payment of invoice {} succeeded.", invoice.id);
                mark_as_paid(&mut invoice);
            } else {
                println!("Payment of invoice {} failed.", invoice.id);
                mark_as_failed(&mut invoice);
            }
        }
    }
}
```

This is sequential and slow. So, obviously, you'd like to parallelize it to get it done faster. You remember your favourite course, ECE 459, and several techniques from it, so you know that you could parallelize this.

More parallelism is not always better, however, because there is the possibility of being rate limited by the third-party service. They limit the rate at which you can call their service to prevent denial-of-service attacks. If the limit

is reached, they start rejecting requests that exceed that limit with an error, but you can retry again at a slower pace. If a request is rejected, `try_to_pay()` (which returns a `Result` type) will return an `Err` where the error code indicates the reason is the rate limit. An additional complication is that they don't tell you what the limit is, because they're just that kind of service.

You are now in a position where you are mentoring a more junior person on your team. Thus, rather than doing the code change yourself, you need to describe a design. The design should maximize the rate of completion of (trying to) pay invoices—but it must reduce the level of parallelism (and retry rejected invoices) if we are being rate limited.

Write down a description of your design. Your description should be sufficiently detailed for a competent junior person to complete the implementation, but not so precise that you are doing it for them. To be more specific, aim for a not-so-detailed pseudocode-level description. Include some analysis of any trade-offs in your design and give your reasoning for important design decisions.

3 A Picture is Worth a Thousand Wordles [10 marks]

You have probably heard of Wordle, or at least, you've probably wondered why so many people are posting green boxes on their Twitter feeds all the time. In short, it's a word game where you have six attempts to guess a five-letter word. You get some colour-coded feedback about the letters you entered: grey for letters that aren't in the word, yellow for letters that are in the word but in the wrong place, and green for letters that are correct.

You are provided no information about today's word when you start the puzzle, and therefore you need to just guess a word and proceed from there. While you could write code to solve the puzzle for you entirely, that takes away some of the fun. So maybe a better idea that is to ask the computer to help choose an optimal starting word. How? Letters appear with different frequencies in the English language in general (e.g., the letter E appears much more often than the letter Z). We can be even more accurate by searching a specific list of five-letter words. For simplicity, we'll restrict this to lower-case letters only.

The provided program takes, as an argument, the name of a file containing five letter words; it will analyze that file to find the frequency of each letter in the standard 26-letter English-language alphabet. You should first count up the frequency of each letter: if the letter A appears in the file 5000 times, the frequency of A is 5000. (Note that the provided sample input files contain a header row where it says `[word]`; this is ignored in parsing). **This is a parallelizable task, so you should modify the code to use 4 threads to analyze letter frequency.**

After analyzing the letter frequency, the code assigns a "rank" to each letter – if A is the most common letter, its rank will be 26; if B is the least common its rank will be 1. Then, score each word based on the ranks of the letters. If A appears in the word, that increases its score by its rank (e.g., 26). Duplicate letters do not count: if there are two As in the word, the second A does not increase the score at all. **Scoring the words is also a parallelizable task, so you should again use 4 threads for this task.**

Finally, print out the highest score (see sample output below) and a comma-separated list of the word(s) with the highest score. If the top-ranked letters are N E S R A, your program will output all valid anagrams of that word. A sample run of the program looks like:

```
There are 500 words to evaluate.
Max score is: 42.
Suggestion(s): nears, earns, saner, snare
```

Complete the starter code provided in the `q3` directory to implement the functionality as described above, respecting the output format as in the example.

4 We Built This City on CUDA [25 marks]

The city of Bielefeld wants to decide where to place segregated bike lanes. You are given a 2-dimensional array P, where each element is a grid square recording the popularity of that part of the city. The city is hilly; you are given a 4-dimensional array A such that $A[x_0][y_0][dx][dy]$ represents the difficulty of going from (x_0, y_0) to $(x_0 + dx - 1, y_0 + dy - 1)$. Here, dx and dy are each in $[0, 2]$ so that we can record difficulties for adjacent squares. All data entries are CUDA `floats` between 0 and 1. Array P contains data from indices $[1][1]$ through [SIZE][SIZE] and a border of 0s around the data elements.

Your task is to use CUDA to find the index of the largest weighted sum

$$\begin{aligned} P[x][y] \quad &+ \quad A[x][y][1][0]P[x][y-1] + A[x][y][0][1]P[x-1][y] \\ &+ \quad A[x][y][1][2]P[x][y+1] + A[x][y][2][1]P[x+1][y]. \end{aligned}$$

My solution uses two kernels: one to compute weighted sums, and one to find potential max indices. You have to do most of the work of finding the max index in CUDA, but you are allowed to do a small amount of final reduction on the CPU. We are not going to evaluate how well you use CUDA, but you have to use it.

Arrays P and A are available on host side but you'll need to make them available to the kernels. We've also provided macros `twoD` and `fourD` in the CUDA code.

Be careful: I've defined SIZE to be 1022, but that doesn't include the border. For many purposes you will want to use (SIZE + 2) (and you are going to be better off if you always put that in parentheses).

Complete the starter code provided in the `q4` directory to carry out the described computation. Your program must print out the twoD value for the winning grid square.

5 Profiling [20 marks]

In your `q5` directory you will find a copy of a text fuzzy search engine, simsearch. Just like Assignment 4, it is set up to create flamegraphs on a sample HTML file when you run "make". Your task is to profile simsearch.

Identifying targets (5 marks). Create the flamegraph and investigate it. There are 4 top-level parts of the execution which account for more than 5% of the runtime on my computer (i.e. methods called by `main`); even if it's less than 5% on your system, consider the 4 biggest ones. Calculate how much speedup you could potentially get by reducing each of these parts' runtimes to 0 (in isolation).

Challenges (5 marks). It's not quite obvious to me how one would speed up this execution. Consider the largest part of the execution. Assume that you control simsearch but no other code. Write down one challenge that makes it hard to speed up the largest part; it may be technical or nontechnical. But, maybe you really need to speed it up anyway. How could you overcome the challenge?

Output time (10 marks.) This program is spending a lot of time preparing output. Implement a change that still allows you to verify the output, but that spends much less time preparing output. To be clear: the modified program can and probably must output something else. Briefly describe the change in a CHANGES file in your `q5` directory (this can be very brief!), and include the machine that you ran it on and before-and-after times. How does this compare to your speedup estimate in the previous part? Don't forget to commit and push the change!

Constraints: You can't hardcode any computation results. You can use any libraries that you want.