ECE 459 Winter 2021 Final Examination

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. Questions will not be answered 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. 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; keeping it brief is preferred.

Programming Questions (Q3 - Q6)

1. You are allowed to modify the code and the makefiles 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 sever 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) (Parallelization.) Give an example of (1) a computation that scales just-about-perfectly with the number of CPUs you give it, and (2) one that can't use more CPUs without restructuring. Name a reason that some programs are in between (1) and (2), getting decent but not perfect speedup.

(b) (Queueing Theory.) It costs money to run servers in the cloud, so you might start up instances only when you need them and turn them off when they are not in use. Suppose that when a job arrives, you start an instance for it (assume setup costs are free), and when the job is done the server is immediately terminated (with no shutdown costs). Assume also that you have set no limit on the number of servers that can exist concurrently and therefore there will not be any queueing. When a server is on, it costs your company $0.001 (one tenth of a cent) per second in computing cost. Jobs arrive at a rate of 10 per second. Service times for those jobs are uniformly distributed ranging from 1 to 9 seconds. Calculate the average spend per second of this system.

(c) (Reduced-resource computation.) You have an array of 1 billion integers and you have to compute the sum. But you're in a rush. You vaguely remember hearing something about loop perforation: you can compute the average over (say the first) half of the array elements instead, saving you a bunch of time. (1) Give an example of where this approach goes badly wrong. (2) What is one data source for the array where this would be likely to go well?

(d) (Hashing.) A colleague tells you that he's sped up the login time for users by 50% by switching from the bcrypt hashing algorithm to md5. Explain why this is a bad idea.

(e) (GPUs.) This is slow.

```
// even the name is slow!
__global__ void slow(float *a, float *b) {
    // i.e. mod 2
    if ((blockIdx.x & 1) == 0) {
        expensive_operation_one(a, blockIdx.x);
    } else {
        expensive_operation_two(b, blockIdx.x);
    }
}
```

Why is this slow? How could you redesign your program to make it faster?

(f) (Compiler optimizations.) It's possible to direct the compiler to inline a function using the `#[inline(always)]` macro. Describe how you would decide where to use it, and how to test if this change was beneficial.

(g) (Boxing.) In Lecture 19 I mentioned that one optimization was to reduce the size of a type by boxing one of its components. How can this backfire, i.e. result in slower runtimes? (For instance: "X is faster but Y is slower and it turns out that, on our benchmarks, Y happens more often"; fill in X and Y.)

(h) (Rust/threads.) As a part of the strategy of "do less work", we have sometimes recommended the strategy of cancelling threads. Is this strategy easier or more difficult in Rust as compared to C? (Tip: if you are a non-ECE/SE student and haven't programmed in C before, check the pthreads pdf in the course repo).

(i) (I/O.) You are doing some computation that involves a long pipeline of (independent, heterogeneous) tasks. You control all of these tasks, i.e. you can rewrite them as needed. This computation takes longer than you'd like. You profile your computation and find that many of the tasks are spending a lot of time doing I/O. How can you speed up your computation? Of course, there are no free lunches; briefly discuss one potential barrier to, or disadvantage of, your improvement.

(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 Is it Atomic? [5 marks]

Write down a function which calls `AtomicI32::compare_and_swap` and a function running in another thread which shares that `AtomicI32`. Write the possible results from your code as written, as well as one possible result if the CAS operation was not, in fact, atomic.

3 After Election Day [20 marks]

An important part of running an election is counting the collected votes. In the `q3` directory you will find some code from `tallystick` which tallies votes. I've added a `q3/src/main.rs` file which is a simple test driver. It manufactures 10 million weighted votes and adds them to a tally.

Your task is to parallelize this code to use 4 threads. As in Assignment 4, you are not to trivialize the problem. In this case, that means that you have to have 10 million calls to the `add` function and you have to produce the same answer; you can't hard-code the final results in the vector. You may, however, create more `tally` objects and split the iteration into different threads, and you may start the iterations partway through the 10 million votes, hardcoding the start point (though there are other valid approaches as well).

On ecetesla0, my solution achieves a speedup of 2.87× over the original code (and 3.1× over the code with 1 thread and locks). For full marks, achieve a speedup of 2.5× over the original code.

4 Election Day [20 marks]

Continuing on the voting theme... Before elections, professional polling companies ask eligible voters whom they will vote for in the upcoming election. Sophisticated companies apply an adjustment based on how likely a person is to actually vote, because not everyone who is eligible will vote. Even though they should.

Our simplified simulation will involve an election where a voter's choices are A and B. We have data about approximately 100 000 voters and we will run 500 000 simulations. At the end of all the simulations, your program will output to standard out the percentage chances for each candidate to win.

A voter is represented by three numbers in the provided CSV file. We read this into a collection of `Voter` (`float3` in the kernel). The array of voters is your input to the simulation. The simulation takes place in the CUDA kernel. One work item is one pretend run of an election. It returns an `ElectionOutcome` (`uint2` in the kernel). A work item does the following steps:

For each voter, generate a pseudorandom number with the provided `pseudorandom()` function. The generator function produces consistent results so that runs of your program are reproducible (this may help with debugging). The argument to this function should be `(provided random seed called SEED + index of the current work item + the current run number [iterator of the loop in the kernel])`. It returns a pseudorandom value P.

Using that value for P, the current voter `v`: (1) votes for A if $P <$ `v.x`, they vote for B if `v.x` $<= P <$ (`v.x + v.y`) and they vote for nobody if $P >=$ (`v.x + v.y`). `v.z` is just used to numerically represent the chance that they do not vote. Each voter's values of `x, y, z` sum up to 1.0.

An election outcome has an `x` field representing the number of votes for candidate A and the `y` field representing the number of votes for candidate B. This is the output from the CUDA kernel.

Determining who won each election simulation is done in CPU code, and so is printing the results. Both of those are provided for you. Complete your kernel code in `q3/kernel/kernel.cu` and the host (CPU) code in `q3/src/cuda.rs`. You shouldn't need to modify `main.rs` but you can if you wish.

To get full marks in this question, you need to use the GPU efficiently.

5 Profiling [15 marks]

In your `q5` directory you will find a copy of a toy web rendering engine, Robinson[1]. Just like Assignment 4, it is set up to create flamegraphs on a sample HTML file when you run "make" and hyperfine with "make hyperfine". Your task is to profile Robinson and implement a change that speeds it up by at least 10%. (I (PL) made a two-line change that improves running time by 27% on ecetesla0).

Identifying targets (2 marks). Create the flamegraph and investigate it. Which top-level part of the execution (i.e. method called by `main`) would you focus on first, and why?

Improvement strategies (4 marks). Based on the profiling data, describe two different strategies you might use to speed up Robinson. Specifically describe each of these strategies—write down which method(s) you would change, a description (in words) of the change you would make, and a potential drawback of that change. For example, "in method html::parse, I would create the dom::elem twice instead of once. This would make the program run twice as fast! (or slow?)"

Implementing an improvement (9 marks.) Implement a code change that speeds up Robinson by at least 10%, briefly describe it in your exam writeup (this can be very brief!), and include the machine that you ran it on and before-and-after times. Your change must generate the same output.png file as before, and you can't hardcode the input. You can use any libraries that you want. Don't forget to commit the change!

[1] You totally don't need to understand how it works and you don't really have time to, but here's a description: `https://limpet.net/mbrubeck/2014/08/08/toy-layout-engine-1.html`.

6 Speculation of the non-stock-market variety [10 marks]

Suppose you're working at a company that charges customers for their product via a simple billing service. It generates invoices and tries to pay them. For some customers, VAT (Value-Added Tax, like GST) applies. There are a bunch of rules and laws about this, but for the sake of simplicity we will say that you must charge the tax unless the customer has a valid VAT number. Fortunately, the government provides a service that checks the VAT number and tells you if it's valid or not.

The program does the following: generate the invoices for each company. For each invoice, check the VAT number to see if it's valid; re-issue the invoice if needed. Then it will pay the invoice. If successfully paid, it may trigger check for fraud. Finally, the updated invoice is published (sent to an external system).

This is a (simple) program that permits speculation in a couple places. Modify the provided starter code to apply speculation to the two places described below.

VAT (5 marks). Sometimes a provided VAT number is not valid. If that's the case, you need to re-issue the invoice with taxes charged on it. To speculate, prepare the invoice with VAT while you are checking the VAT number for the company.

Fraud (5 marks). If an invoice is above a certain amount, it needs to be analyzed for fraud. If fraud is found, the fraud result is uploaded. You can calculate the fraud result in parallel with paying and publishing the invoice; publish it only if the invoice was successfully paid and the amount above the fraud check minimum.