ECE 459 Winter 2020 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 to the written question dropbox, and a zip file with your code to the code question dropbox.
6. Be sure to submit your files on time.

Written Questions

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

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 OpenCL) machines.
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 OpenCL problems, so try using `eceubuntu` when the GPU is not needed.
7. Remember to complete the `README.txt` to inform the marker about the sever you did each question on.
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 [16 marks]

Answer these questions using at most three sentences. Each question is worth 2 points.

(a) Helgrind reports lock ordering errors on acquisition of the lock, but does not do so on releasing the locks. Explain why.

(b) What problem might be observed with this code and how do you fix it ?

```
void transfer_funds( account* restrict sender, account* restrict recipient, double amount) {
  pthread_mutex_lock( sender->lock );
  sender->balance -= amount;
  pthread_mutex_unlock( sender->lock );
  pthread_mutex_lock( recipient->lock );
  recipient->balance += amount;
  pthread_mutex_unlock( recipient->lock );
}
```

(c) In our discussion of cache coherence, we talked about *false sharing*. Is this phenomenon likely to occur with stack allocated memory or heap allocated memory? Explain briefly.

(d) Is the `memset` function (it sets every byte of a section of memory to a specific value) a good candidate for the compiler to generate SIMD instructions for? Explain your answer.

(e) Show an example of a reordering of operations that could happen under weak consistency, but not sequential consistency.

(f) Explain why alias analysis is difficult for the compiler in C.

(g) What probability distribution is appropriate for modelling the scenario of how many users give your app a review in the store? Explain your answer.

(h) One of your colleagues opposes using containers for delivering your app for security reasons. Does this colleague have a good argument? Explain.

2 Not-As-Short Answer [24 marks total]

2.1 Rainbow Table [6 marks]

Imagine you have a non-evil reason for cracking a password, and rainbow tables to go with the hashing function `secure_hash`. You have the following functions available:

```
function secure_hash( string input ) : returns string
function clever_reduce( string input ) : returns string
function get_chain_number ( rainbow_table table, string hash ) : returns int
function get_starting_plaintext( rainbow_table table, int chain_number ) : returns string
```

The `get_chain` function returns the ID number of the chain if found, and -1 if it is not found.

Using the functions above, write a pseudocode implementation of the routine to find a plaintext.

```
function find_plaintext( rainbow_table table, string hash ) : returns string
```

2.2 Rust [4 marks]

Memory. Write down an example of C code demonstrating a run-time memory error that would be prevented at compile-time by the Rust compiler. Explain why the error does not occur in Rust, referencing the language features that prevent the issue.

Concurrency. Write down an example of C code demonstrating a run-time concurrency error that would be prevented at compile-time by the Rust compiler. Explain why the error does not occur in Rust, referencing the language features that prevent the issue

2.3 Trading Accuracy for Time [5 marks]

In lecture, we discussed the problem of how Google Maps (or any similar software) decides on the route suggestions it offers when giving directions from point A to point B. This is only an interesting problem when there are actually multiple reasonable options, as is frequently the case when driving in Toronto which has a partial grid system. Brute forcing all possibilities is not feasible (given the time frame). Come up with three (3) strategies as to how Google might be doing this, explaining why it is a valid approach. Then select which one you think is best, and explain why you chose that.

2.4 Queueing Theory [5 marks]

Credit: Mor Harchol-Balter, *Performance Modeling and Design of Computer Systems*

Response Times [2 marks] Your webservice receives the following requests. Every even-numbered second the system receives 2 requests. Every odd-numbered second, it receives 1 request. If the average time to complete a request is 6 seconds, how many requests will be in progress, on average? Justify your answer using queueing theory.

The Fight Does Not Go Well, *Enterprise* [3 marks] After spending months carefully building a **closed** batch data storage system, Robin comes to see their boss with the following description and measurements: The MPL for the system is fixed at 19 jobs. Robin explains that 90% of jobs find the data they need in the cache, and hence their expected response time is only 1 second. However, 10% end up having to go to the database, where their expected response time is 10 seconds. Robin's boss asks one question: "How many jobs do you see on average at the database?" When Robin answers "5," the boss says that this does not make sense. What is wrong?

2.5 It's a lock-free country [4 marks]

The gcc atomic definitions include the following:

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)
```

The documentation says that `type` can be any integral scalar or pointer type that is 1, 2, 4 or 8 bytes in length. You can ignore the last argument (the `...`) in your usage. Use these to complete the implementation of the skeleton code below:

```
typedef struct {
  void * data;
  void * next;
} s_node;

typedef struct {
  s_node * top;
} lf_stack;
```

```
void init( lf_stack * s ) {
   top = NULL;
}

void push( lf_stack * s, s_node * n ) {

}

/* Returns NULL if the queue is empty */
s_node * pop( lf_stack * s ) {

}
```

3 Profiling [10 marks]

The `q3` folder has a program that does edge detection. This is a form of image processing. The program explains its parameters if you invoke it without arguments, but a sample invocation is: `bin/canny_edge q3/0496_rocks.pgm 0.5 0.8 0.2`. Use the sample image provided and profile this application; you can test with different parameters to get a good data set.

Use `perf` to analyze this program. Write a brief report about what you learned, including: (1) what you've learned about the program in terms of where the time is going, and (2) if the profiler told you any lies (justify your answer of yes or no). Your explanation should be about 1 page, including an excerpt of the output of `perf` to support your conclusions.

Add your report as part of your submitted document alongside the answers to questions 1 and 2.

4 OpenMP [10 marks]

Use the command "`OMP_NUM_THREADS=4;export OMP_NUM_THREADS`" to set 4 threads.

The Riemann Hypothesis is an important open question in pure mathematics about where Riemann zeta function may have zeros. This has some applications in math including the distribution of prime numbers.

I've provided some code to compute the Riemann zeta function at various points in `q4/zeta.c`. This code takes a number of parameters: a number of iterations `r`, a number of points `N`, and a starting point `d`. Benchmark your sequential program such that the execution time is approximately 10 seconds.

Using OpenMP pragmas and the minor code structure changes that you'll need to support them, parallelize the program in `q4/zeta_omp.c`. Benchmark your manual parallelization with OpenMP and calculate the speedup over the sequential version. Use `hyperfine` to collect your timing data.

Goal speedup: 3.9

5 Taking a Little Trip [25 marks total]

The `q5` program is a solver that generates a travel route given a map and a start and end location. It tries many possibilities and chooses the best one it has found. It takes an input file and the amount of time to execute for. The more iterations, the better a route it is likely to find, where better means less distance travelled. The program is invoked like: `bin/solver q5/tsps/berlin52.tsp -s 10` for 10 seconds of solving the `berlin52.tsp` file. If it helps you, you can also invoke the program with a `-i` argument followed by a number of iterations, allowing you to keep iterations constant but measure execution time. The Makefile also compiles the `solver_fast` executable, the one with your modifications.

Running the solver produces an output file `berlin52.tour`; only the third line is interesting. It contains the number of iterations and the distance and looks like `COMMENT: 1000 iterations, 9775 distance`. A better implementation will execute more iterations in the same amount of time, and is more likely to achieve a better result (less distance).

Before you begin, take a baseline measurement for how many iterations you can accomplish in 10 seconds with `solver`. Write this down for later. Your goal is to modify the `gatsp_fast.cpp` to get speedup in `solver_fast`. You may use any techniques we have discussed in the class, such as threads, OpenMP, or modifying the single thread performance. If you choose multiple threads or OpenMP, use **only 4 threads**. Run `solver_fast` with a runtime of 10 seconds to determine how many iterations your modified program has completed. To determine your progress, divide the number of iterations from `solver_fast` by the number of iterations from `solver`.

You may not modify `solver.cpp` or its interface with `gatsp`.

Goal increase in number of iterations: $2.5\times$.

6 Election Day [35 marks]

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 100 000 voters and we will run 100 000 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. Read this into a collection of `cl_float3` (`float3` in the kernel). The array of voters is your input to the OpenCL kernel, alongside the number of simulations to run, and the provided random seed.

The simulation takes place in the OpenCL kernel. One single run of a user's behaviour is not enough work for a single work item, so the work item should perform `100 000` runs. Each run involves generating a random number with the provided `rand()` function. The argument to this function should be `(provided random seed + get_global_id(0) + the current run number)`. It returns a random 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 is represented by a `cl_uint2` (`uint2` in the kernel) where the `x` field represents the number of votes for candidate A and the `y` field represents the number of votes for candidate B. This is the output from the OpenCL kernel.

Determining who won each election simulation is done in CPU code, and so is printing the results. Complete your kernel code in `q6/election.cl` and the host (CPU) code in `q6/election.cpp`.