Final Project Part 2: Threads

You should have completed Part 1 before starting on Part 2.

This portion of the project is about multi-threading. We will be using OpenMP for threading. It's great for parallizing loops.

New Command `--run-by-proxy`

Introduce a brand new option for `runlab`: `--run-by-proxy`!
What does it do? Well, it's just like `--run-git-remotely` except you don't have to commit your code! It just pulls it out of your local directory.
You should consider this a "beta" feature. If it causes problems, I'll have to turn it off.

Turning on OpenMP

To turn on OpenMP, you need to set

```
OPENMP=yes
```

in `config.env`, if it's not set already. This sets some compiler flags to enable it.
OpenMP uses a thread pool – a collection of theads that sit around waiting for work to do. You can set the size of the pool with `--threads` command line argument. Add it to the `IMPL_SEL_ARGS` because we want to test the code with multiple threads:

```
IMPL_SEL_ARGS=--threads 8
```

8 is a reasonable choice because there are 8 thread contexts spread across 4 physical cores in our system. It's conceivable that 4 might give better performance (for reasons you'll hear about in 142 when you discuss SMT).

Threads Make Everything Harder

Threads and concurrency make programming and program behavior much more complex. We could easily spend a whole quarter (or year) studying it. We are using OpenMP because it simplifies things significantly and it's a good match for CNNs. However, the tools we are using all have some limitations with respect to threads.

Non-Determinism

Multithreading introduces the possibility for a new and really annoying bugs. If you fail to coordinate your threads correctly, it can give rise to non-deterministic bugs that only occur sporadically.

This means that your program can run correctly one time and incorrectly next. If this occurs, you are not losing your mind (although it may feel that way).

These intermittent failures are often always due to two threads sharing data without using a lock to ensure that only one thread is accessing the data at one time.

Many Tools Don't Fully Support Threads

First, `gprof` doesn't work on with multi-threaded programs.
Second, Moneta's cache model is not multithread-aware, so the color coding for the traces for hits vs. misses and the hit and miss rates are not accurate.

Moneta will also show more threads that you might be expecting. OpenMP threads seem to be Thread 0 and 13 and above. If you see some threads that don't seem to be doing anything, that's not surprising or concerning.

Finally, our performance counting code only collects data for one thread. For OpenMP code this is ok: all the threads do basically the same thing. But you'll notice, for instance, that if a loop runs in 4 threads, the measure instruction count will go down by ~1/4 (assuming multi-threading didn't add a lot of overhead).

Amdahl's Law for Multi-threaded Code

Gprof's non-support of multi-threaded code presents a particular problem: You need to know where your code is spending the most time, so you can focus your optimization efforts. How else will Amdahl be appeased?

For this project, we can get a decent approximation by running running each function in each layer once and adding up the execution time. This number should approximate the total runtime of `train_model`, and you can use the latencies of each function to guage how important it is. To make this work, you'll need to pass `--scale 0`, so each function runs exactly once.

Tasks To Perform

Parallelize fc_layer_t::calc_grads()

To get you started, we will walk you through the process for parallelizing `fc_layer_t::calc_grads()`. Please see the slides in the lab repo for more details. They contain detailed description of how the code works.
Unlike the previous labs, we have provided a baseline implementation in `opt_cnn.hpp`.
Run `runlab --no-validate`. It should finish and in the output, you'll see

```
...
[  PASSED  ] 21 tests.
```

Which means that your implementation matches the result of the baseline (which is no surprise because you have not edited baseline).

These tests are your best friend, since they provide a quick and easy way of telling whether your code is correct. `runlab` runs the tests every time, and if you the last line shows any failures, you should look at `regressions.out` for a full report.
First, replicate the structure in `activate()` to let you select among several implementations of `calc_grads`. Make two copies of the `calc_grads` starter code. One you'll leave alone and use as starting point for more copies (see below). The other you'll optimize.
Set it up so that `--param1 1` will run the baseline version and `--param1 2` will run the new version.
Then change `OMP_NUM_THREADS` to 2 in `config.env`.

Version 1: `nn`-Loop

Modify the code to add multithreading to the `nn` loop and run it again. You can do this by adding `#pragma omp parallel for` on the line before the `nn` for loop. When your code finishes running, you will notice that you failed multiple regression tests. This is because by parallelizing the `nn` loop, multiple threads attempt to write to the same location in `grads_out`.
We will fix this in two stages:

Stage 1 Add a local tensor the same size as `grads_out` at the top of the `nn` for loop. You can see an example of this in `example/stabilize.cpp` line 338. The example creates a tensor of type double with the same size as `output`. You will do the same except the tensor size will be the same size as `grads_out`. Do not forget to clear it

just like the example does on line 339. Then, in the inner most loop (the `i` loop), change `grads_out` to be your new local tensor that each thread will create.
This enables each thread to accumulate their results locally. Thereby eliminating the race condition causing errors. However, we now need to combine the results from each thread.

Stage 2 At the bottom of the `nn` for loop, add a critical section and create two nested for loops to loop through `out.size.b` and `grads_out.size.x`. Notice that we don't loop through `out.size.x` as well. This is because we only need to accumulate the results into `grads_out` and `n` is not used to index into `grads_out`.
Inside the nested for loop you just created, accumulate the results of each thread (stored in their local tensors you creaded in stage 1) into `grads_out`. This will look very similar to `example/stabilize.cpp` lines 361 - 369.
Once you have made your changes, run the code locally and verify that you pass all 21 regression tests. If you do not pass, refer back to the lecture slides, discussion slides, example in `exmaple/stabilize.cpp` lines 330 - 372, and help from the staff during office hours or lab hours. Once you have verified that your code is correct and passes the regression tests, submit to the autograder. You will want to save the resulting benchmark.csv file for the worksheet.

Version 2: `b`-Loop

Make another copy of the starter code. Rename it, and add it to the `switch` statement so you can select with `--param1 3`. We are going to apply a different optimization.
Modify the code to add multithreading to the `b` loop and run it again. You can do this by adding `#pragma omp parallel for` on the line before the `b` for loop. You will notice that you passed the regressions test! There is no need to fix any race condition here as each thread is accumulating its result into a different address of grads_out.

Version 3: `n`-Loop

Same thing again: make another copy of the baseline code. This time make it selectable with `--param1 4`.
Modify the code to add multithreading to the `n` loop and run it again. You can do this by adding `#pragma omp parallel for` on the line before the `n` for loop. When your code finishes running, you will notice that you failed multiple regression tests. This is because by parallelizing the `n` loop, multiple threads attempt to write to the same location in `grads_out`.
We will fix this in two stages:

Stage 1 Add a local tensor the same size as `grads_out` at the top of the `n` for loop. You can see an example of this in `exmaple/stabilize.cpp` line 338. The example

creates a tensor of type double with the same size as `output`. You will do the same except the tensor size will be the same size as `grads_out`. Do not forget to clear it just like the example does on line 339. Then, in the inner most loop (the `i` loop), change `grads_out` to be your new local tensor that each thread will create.
This enables each thread to accumulate their results locally. Thereby eliminating the race condition causing errors. However, we now need to combine the results from each thread.

Stage 2 At the bottom of the `n` for loop, add a critical section and create a for loops to loop through `grads_out.size.x`. Notice that we don't loop through `out.size.x` or `out.size.b` as well. We exclude `out.size.x` because we only need to accumulate the results into `grads_out` and `n` is not used to index into `grads_out`. We exclude `out.size.b` because we only need to accumulate the result that the threads were individually working on, and they were all already working on the same `b` because we multithreaded the `n` loop, which is inside the `b` loop.
Inside the for loop you just created, accumulate the results of each thread (stored in their local tensors you creaded in stage 1) into `grads_out`. This will look very similar to `exmaple/stabilize.cpp` lines 361 - 369.

Version 5: `i`-Loop

One more time. This time use `--param1 5`.
Modify the code to add multithreading to the `i` loop and run it again. You can do this by adding `#pragma omp parallel for` on the line before the `i` for loop. You will notice that you passed the regressions test! There is no need to fix any race condition here as each thread is accumulating its result into a different address of grads_out.

Compare Performance

Let's see which loop we should parallelize. You should be able to measure the performance of all five implementations with one call to `--run-git-remotely`.
In `config.env`, set

```
IMPL_SEL_ARGS=--param1-name impl --param1 1 2 3 4 5 --threads 1
CMD_LINE_ARGS=--test-layer 14 --function calc_grads  --stat-set PE.cfg --
stat misses=PAPI_L1_DCM --engine papi --calc TIC=IC*omp_threads --calc
Tmisses=misses*omp_threads
```

This will run all 5 implementations of `calc_grads` with 1 thread on layer 14 (the largest `fc_layer_t`).

Starting with `--stat-set PE.cfg`, it also sets up some performance counters to measure the components of the performance equation and cache performance. The data can be a little confusing, so here's what the columns mean. The first 8 are our focus:

Column name	**Meaning**
misses	the number of L1 cache misses (for one thread)
IC	Instruction Count (for one thread)
TIC	Total instructions (all threads), `IC * omp_thread`. If you set `--threads` > 1 and don't use OpenMP, this will be inaccurate.
Tmisses	Total missess (all threads).
CT	The actual cycle time (cycles/ET). This might not be 1/MHz due to power throttling for.
CPI	Cycles per Instructions
ET	Execution time
omp_threads	Number of threads
cycles	Actual clock cycles (for one thread)
runtime	Same as ET
PAPI_REF_CYC	"Reference clock" cyles. Ignore this.
MHz	The clock rate we asked for.
IPC	1/CPI

Commit everything. Then do

```
runlab --run-git-remotely -- make cnn.csv
```

and you should get something like this:

```
dataset|training_inputs_count|omp_threads|impl|param2|param3|param4|full_name                          |function  |layer|layer_type|reps|
-------|----------------------|-----------|----|------|------|------|-----------------------------------|----------|-----|----------|----|
mininet|4.0                   |1.0        |1.0 |1.0   |1.0   |1.0   |layer[14]->calc_grads  fc_layer_t|calc_grads|14.0 |fc_layer_t|5.0 |
mininet|4.0                   |1.0        |2.0 |1.0   |1.0   |1.0   |layer[14]->calc_grads  fc_layer_t|calc_grads|14.0 |fc_layer_t|5.0 |
mininet|4.0                   |1.0        |3.0 |1.0   |1.0   |1.0   |layer[14]->calc_grads  fc_layer_t|calc_grads|14.0 |fc_layer_t|5.0 |
mininet|4.0                   |1.0        |4.0 |1.0   |1.0   |1.0   |layer[14]->calc_grads  fc_layer_t|calc_grads|14.0 |fc_layer_t|5.0 |
mininet|4.0                   |1.0        |5.0 |1.0   |1.0   |1.0   |layer[14]->calc_grads  fc_layer_t|calc_grads|14.0 |fc_layer_t|5.0 |
```

P1 (1pt): Which implementation provide the best performance? The worst?

```
Best OMP implementation:

Worst OMP implementation:
```

Multiple threads

The data above shows that adding multithreading support degrades performance, but we should be able to recover it with threads!

Modify the `./cnn.exe` command line above to run with different numbers of threads using `--threads 1 2 4`. You'll get a bunch more data:

dataset	training_inputs_count	omp_threads	impl	param2	param3	param4	full_name	function	layer	layer_type	reps	IC	cycles	runtime	More	miss
mininet	4.0	1.0	1.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.35e+09	8.86e+08	0.268	
mininet	4.0	1.0	2.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.65e+09	1.01e+09	0.307	
mininet	4.0	1.0	3.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.36e+09	8.8e+08	0.272	
mininet	4.0	1.0	4.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	2.59e+09	1.42e+09	0.441	
mininet	4.0	1.0	5.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.43e+09	8.54e+08	0.297	
mininet	4.0	2.0	1.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.35e+09	8.83e+08	0.269	
mininet	4.0	2.0	2.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	8.29e+08	5.39e+08	0.169	
mininet	4.0	2.0	3.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	6.84e+08	6.18e+08	0.196	
mininet	4.0	2.0	4.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.31e+09	1.16e+09	0.367	
mininet	4.0	2.0	5.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	7.41e+08	6.58e+08	0.23	
mininet	4.0	4.0	1.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.35e+09	8.8e+08	0.267	
mininet	4.0	4.0	2.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	4.19e+08	3.05e+08	0.0961	
mininet	4.0	4.0	3.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	3.48e+08	4.79e+08	0.154	
mininet	4.0	4.0	4.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	6.94e+08	1.14e+09	0.363	
mininet	4.0	4.0	5.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	4.15e+08	4.42e+08	0.169	
mininet	4.0	8.0	1.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	1.35e+09	8.82e+08	0.268	
mininet	4.0	8.0	2.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	2.14e+08	2.76e+08	0.0873	
mininet	4.0	8.0	3.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	3.53e+08	7.52e+08	0.24	
mininet	4.0	8.0	4.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	7.12e+08	1.62e+09	0.523	
mininet	4.0	8.0	5.0	1.0	1.0	1.0	layer[14]->calc_grads	fc_layer_t	calc_grads	14.0	fc_layer_t	5.0	2.69e+08	4.63e+08	0.23	

NOTE The statistics are *per thread*. Hence you'll notice that the `IC` values drops by roughly 1/2 between 1 thread and 2.

There are several interesting things about this data. The first is that performance does not decrease monotonically with thread count. The other is that the different implementations have very different performance. Import the data into your favorite graph plotting program (e.g. Excel or Google Sheets) and answer the following:

P2 (10pts): Draw 5 graphs, one for each of the 5 implementations. Each one should plot the normalized values for the first 7 values in the data you collected above (y-axis) vs. thread count (x-axis). The graphs, their axes, and their legends should be clearly labeled (e.g., 'Version 1 – NN'). I'll do an example of how to do this efficiently in class.

```
The graphs.
```

P3 (1pt): Which implementation and thread count provides the best performance?

```
Best implementation:

Best Thread count:
```

Generate a Moneta trace for each implementation running with 4 threads. You can use the same command line options you used above except:

1. Set `--scale 0` to `cnn.exe` (i.e., after the `--`) so the function executes exactly once.
2. Pass `--main none` to `mtrace` (i.e., before the `--`) so `mtrace` won't start tracing until we call `START_TRACE()`.
3. Pass `--trace <name>` to `mtrace` with a different `<name>` for each trace. Use a descriptive name that well let you tell which trace is for which implementation. When you specify `<name>`, you are actually only providing an initial name. `NEW_TRACE()` sets a more descriptive name in `opt_cnn.hpp`
4. Run one implementation at a time.
5. Only run for 4 threads.
6. You might want to pass `--stats-file none.csv` to `cnn.exe` so your other `.csv` files don't get overwritten.

You should end up with 5 traces. They will all terminate early after collecting several million memory operations. This is fine.

Several of these questions ask you to estimate the size of different regions of data. These number just need to be approximate. A good way to measure them is to compare them to the green scale bar that appears at the lower-left of the Moneta trace window. It's the size of the L1 cache (64KB in our case).

P4 (2pt): Consider Version 3 (b-loop). Use the Moneta trace to determine approximately how many KBs of `grads_out` each thread accesses and roughly how many times it accesses those bytes. Provide and label one screen capture showing how you arrived at these values.

```
How many KB?

How many updates (approximately, circle one):  1    16   32   64   Too many
                                                                    to count
```

Screen capture

P5 (2pt): Consider Version 3 (b-loop). Provide a screen capture that shows the access patterns of all four threads for the `weights`. Circle a group of accesses performed by one thread. Approximately how large is `weights` tensor in KB? How many times is each element accessed (across all threads)?

How big is weights (KB)?

How many times is each element accessed?

Screen capture

P6 (2pt): How does what you learned in answering P5 and P6 explain the high CPI for Version 3 (b-loop)? (2 sentences max)

P7 (2pts) Consider Version 4 (n-loop), and assume that each thread is running on its own processor. How much of the `weights` tensor does each thread access repeatedly before moving onto more data (i.e., how big is it's working set)? How many times does it accesses each entry of the tensor? Provide and label a Moneta screen capture that supports your conclusions.

```
How big is the working set?

How many accesses/tensor item?

Screen capture
```

P8 (2pts) Consider Version 4 (n-loop). What's the ratio of IC on the baseline to IC on this implementation (version 4) with 1 thread? Paste in a copy of the C++ code that corresponds to the extra instructions.

```
IC(baseline)/IC(n-loop) =

The C++ Code
```

P9 (2pts) Consider Version 5 (i-loop). How much of the `weights` tensor does each thread access repeatedly before moving onto more data (i.e., how big is it's working set)? Give your answer in KB. Provide and label a screen capture illustrating your conclusion.

```
How much at once (Approximate KB)?
```

Screen capture

P10 (2pts) Consider Version 5 (i-loop). How much (in KB) of `grads_out` tensor is does each thread access? Provide and label a screen capture illustrating your conclusion.

```
How much does each thread access (approx KB)?
```

Screen capture

P11 (2pts) Use your answers to P9 and P10 and similar measurements of the trace of Version 1 (baseline) to explain the difference between the number of misses in Version 5 (i-loop) and Version 1 (baseline). Two sentences max.

P12 (5pts) Compare the data for Version 2 (nn-loop) and Version 5 (i-loop) with 4 threads using all three terms of the PE (IC, CPI, and CT). For each term compute `(value for version 5)/(value for version 2)`. Which term explains Version 2's lower ET? What could be one underlying cause? (2 sentences max)

```
IC:

CT:

CPI:

ET:

Why does Version 2 have a lower ET?
```

Go forth and optimize!

There are bunch of loops in the code base just waiting to be parallelized.

Check `README.md` for performance targets for the end of this part of the lab.
Recall the note at the top of this file about Amdahl's Law, multi-threaded code, and how `gprof` can't handle threads.

Tips

- There are many more things to try in this lab than there have been in the earlier labs. This has two implication:

    - Start early.
    - "guess and check" is unlikely to get you a good solution in reasonable. Think carefully about the changes you are making. Thinking takes time. Start early.
    - The autograder servers are going to get very busy near the deadline. Start early.
- Unfortunately, gprof doesn't work on multi-threaded programs. You can comment out `OPENMP=yes` in `config.env` to make gprof work properly.
- OpenMP is a big library. We've covered a bit of it. You're free to use the rest.
- There's lot of resources on the web about OpenMP. Many (or most) of them are bad. This one is pretty good: http://jakascorner.com/blog/. Especially these entries:
    - http://jakascorner.com/blog/2016/04/omp-introduction.html
    - http://jakascorner.com/blog/2016/05/omp-for.html
    - http://jakascorner.com/blog/2016/07/omp-critical.html
    - http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
    - http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html
    - http://jakascorner.com/blog/2016/07/omp-default-none-and-const.html

---

YOU ARE NOT DONE. THERE IS ANOTHER PART (`README3.pdf`)
