Final Project (and Vectors and Threads)

For your final project, you'll be applying optimizations to a larger CNN model based on the same framework we used in Lab 3. There are two new kinds of layers, more loops to target, and more opportunities to optimize. You'll also have some new tools: threads and vectorization.

Altogether, this means that you'll need to apply the following optimizations:

1. Loop reordering
2. Loop tiling
3. Multithreading
4. Vectorization

This lab will be completed on your own.

Check gradescope for due date(s).

FAQ and Updates

Watch here for answers to FAQs and notifications about important updates.

Integrated Worksheet and README

READ THIS CAREFULLY

`README.pdf` is both the lab instructions and your worksheet for the lab. You should use the provided `README.pdf` to fill in your answers for this lab. **You should not generate your own pdf**, since the formatting may not match.
As bugs are found in the lab, we may update `README.md`. We **will not** update `README.pdf` so we can ensure that everyone is using the `README.pdf` to submit answers.
We will maintain a FAQ/important updates section at the top of `README.md` *and* incorporate changes into the body of the document.

Annotating PDFs

We realize that marking up PDFs is kind of terrible, but it's the best solution we've found to the problem of submitting and grading these labs (we have considered, many, many alternatives and none of them work well for all students). To make it as easy as possible, here are some options.

Option 1: Write in the PDF itself using either text-based or freeform (e.g., writing with a tablet) annotation tools.

Option 2: Convert the homework PDF to images and use those images as backgrounds in Word or LaTeX or Google Slides (Google Docs doesn't really support background images). Type on it, then save it as a PDF.

You can try exploring the following resources to edit pdfs to fill in your Homework Solutions.

1. https://www.pdfescape.com/open/
2. https://simplypdf.com/
3. https://www.foxitsoftware.com/

Keeping Your Repo Up-to-Date

Occasionally, there will be changes made to the base repository after the assignment is released. This may include bug fixes and updates to `README.md`. In those cases, you can use the following commands to pull the changes from upstream and merge them into your code.

```
runlab --check-for-updates
runlab --merge-updates
```

Grading

This is a ~3 week lab with two checkpoints.

Your grade for this lab will be based on your completion of the data collection steps described in this document and the completed worksheet.

Part	value
Part 1	23%
Part 2	23%
Part 3	23%
Overall Speedup	31%

The write up for this lab is broken into three parts: `README.pdf`, `README2.pdf`, and `README3.pdf`.
Parts 2 and 3 will be released shortly.

`README.pdf` is due after the first week. `README2.pdf` is due after the second, and `README3.pdf` is due after the third. You can turn any part in late for 75% credit. We will only grade one submission for each part.
Everything is due at the end of week 3 (note that this quarter, this week is short – The due date is on Friday of week 10 instead of Sunday). No late work or extensions will be allowed.

Please check gradescope for exact due dates.

The Leader Boards

There is a leader board set up for this lab. It records the speedup of your code vs the starter code for neural net training. You can use it to guage your progress.

The leader boards do not impact your grade.

Example Code

The `example` directory contains the image stabilization example from the lab lecture slides. You shouldn't (and won't need to) use any of the code from that example for this lab. However, it may be helpful to study the code. It also has another instance of using `--param*` to select implementations, this time using fancier C++ stuff.
The directory also contains `matmul.cpp` which is the the code I used for the iteration space example in class.

Skills to Learn and Practice

1. Using profiling to guide optimization.
2. Using OpenMP for Multithreading
3. Applying loop reordering.
4. Applying loop tiling.
5. Vectorizing loops with OpenMP and GCC Auto-vectorizer
6. Testing code.
7. Quantifying the benefits of optimizations

Software You Will Need

1. A computer with Docker installed (either the cloud docker container via ssh, or your own laptop). See the intro lab for details. You'll need docker image `ucsdnvsl/cse141pp:sp21.180`.
2. The lab for the github classroom assignment for this lab. Find the link on the course home page: https://github.com/CSE141pp/Home/.
3. A PDF annotator/editor to fill out the worksheets.

Tasks to Perform

Inspect The Code

There are several source files in the lab, but you'll only be editing one:

1. `cnn.cpp` – The driver code is in `main()` (mostly data collection and command line parsing). Also, code for creating, training, and running ML models.
2. `opt_cnn.hpp` – A soon-to-be-optimized (by you) version of two CNN primitives.
3. `examples.cpp` – Some sample code showing different ways to vectorize and parallelize code. It's not strictly necessary for the lab, but might be useful.
4. `model.cpp` – This has the `train_model` (which is the benchmark function for the lab) and the code to instanitate the CNN.
5. `parameters.cpp` – holds the global parameters (see below.)
6. `reps.cpp` – holds a table that controls how `--scale` affect the number of repetitions for each function.

You will only be graded on changes made to `opt_cnn.hpp`. Changes to other files will have no effect on the autograder's output. However, you can modify them to add Moneta tags, if you think that would be useful.
The basic flow is like this:

- Execution starts in `cnn.cpp` which loads the input dataset.
- `cnn.cpp` executes the "canary" to verify the machine is performing properly.
- `cnn.cpp` executes one or more experiments depending on the command line options you pass. `./cnn.exe --help` will show the defaults. There's more detail about running multiple experiements below.

You'll find four skeleton classes in `opt_cnn.hpp`. They inherit from the corresponding classes in Canela. Right now, oly `opt_fc_layer_t` has any code in it. For the others are empty, so using them is the same as using original Canela classes.
To optimize them, you should copy the functions from Canela you want to optimize into these classes. Any changes you make will affect the performance and correctness of the tests run by `cnn.cpp`.

Learn a Little About the Model

The model we are using for this lab is a simplified version of AlexNet (https://en.wikipedia.org/wiki/AlexNet), the deep neural network that kickstarted the current mania around deep learning.

AlexNet was built to compete in the ImageNet competion (http://image-net.org/challenges/LSVRC/2016/index) which measure accuracy in classifying 256x256-pixel RGB images. AlexNet was enormous (for the time):

1. It had 60 million internal weights
2. It took two weeks to train it using two big, fast GPUs.
3. If expressed using the layers that Canela provides, it has about 20 layers.

Our version is a bit smaller. It works on 128x128 images and is only 18 layers. It also does not include a few sophisticated optimizations to the learning process.

It is built in `model.cpp`.
The model contains multiple convolution, pooling, and fully-connected layers and the whole model occupies 590MB of DRAM, so make efficient use of caches is very important.

Test Everything

Like last time, get started by checking out the code and checking it locally with

```
runlab --devel
```

This is equivalent to running

```
make default
```

The code will run for a while. On our machine, the starter lab runs for about 40s. Your local machine may be slower or faster.

You'll get a few files:

1. `regression.out` has the report from the regression suite.
2. `benchmark.csv` is the csv file used to measure performance `CMD_LINE_ARGS` has no effect.
3. `cnn.csv` is similar to `benchmark.csv` but `CMD_LINE_ARGS` has its normal effect.
4. `cnn.gprof` and `benchmark.gprof` are not here now, but if you set `GPROF=yes` they will appear.
5. The output will also contain the feedback from the gcc auto-vectorizer module when AUTO_VEC=yes is set in `config.env` (this will show up in `STDOUT.txt` with `--run-git-remotely`)

You can submit it to the autograder with `--run-git-remotely` for good measure, if you want.
You can run

```
runlab --run-git-remotely -- make default
```

To run everything on the remote servers, or

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

or

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

to run one version or the other. For instance, while you are optimizing one function there's not much point in running the data for `benchmark.csv`, and skipping it will save time.

Command Line Options

NOTE This part has changed significantly since the last lab.

Your executable takes a few useful command line options we haven't discussed:

- `--scale <n>` sets the execution time for each tested functions to `n` seconds (approximately). For very short functions, it runs them 10,000 times.
- `--function <function_name>` Collect measurements for a particular function. Use `./cnn.exe --help` (after you've done `make cnn.exe`) to get a list of possible values.

Read the Source

You need to get acquainted with the code you'll be optimizing. The slides from the lab lecture are an important resource here, especially for memory layout of `tensor_t` and how the functions in `fc_layer_t` work.
The baseline version of Canela is in your docker repo in `/course/CSE141pp-SimpleCNN/CNN`. You should read through the commented code in these files (some of these are familiar from prior labs. Nothing significant has changed in these files):

- `tensor_t.hpp`
- `types.hpp`
- `layer_t.hpp`
- `fc_layer_t.hpp`
- `conv_layer_t.hpp`
- `pool_layer_t.hpp`
- `relu_layer_t.hpp`

In each of these files there's some code near the top that has lots of comments. This is the code you should focus on. There's a lot more code down below, but it's all utility functions and debugging support. It's not important to the operation of the library or your optimizations.

The point is not to deeply understand the code at this point. Rather, it's to become acquainted with where the code is. This will make it easier to answer questions about the code that you have later.

Measuring Performance of Particular Functions

There are quite a few functions to tune and optimize in this lab. To make this easier (and less time consuming) the `cnn.exe` executable has built-in support for running and timing specific functions and for getting information about the model.
The option `--describe-model`. This will list some information about the model and exit. For instance, for this lab,

```
./cnn.exe --describe-model
```

Give you something like this (Don't rely on this. It might be out of date. Run it yourself.):

```
Using dataset mnist
IN     (28, 28, 1, 1)
layer[0]  -> (7, 7, 96, 1) 3.2e+02 kB (0.14%) : conv_layer_t(stride=4, kernel_size=11, kernel_count=96, pad=0)
layer[1]  -> (4, 4, 96, 1) 86 kB (0.038%) : pool_layer_t(stride=2, filter_size=3, pad=0)
layer[2]  -> (4, 4, 96, 1) 36 kB (0.016%) : relu_layer_t()
layer[3]  -> (4, 4, 256, 1) 1.4e+04 kB (6.3%) : conv_layer_t(stride=1, kernel_size=5, kernel_count=256, pad=2)
layer[4]  -> (2, 2, 256, 1) 72 kB (0.032%) : pool_layer_t(stride=2, filter_size=3, pad=0)
layer[5]  -> (2, 2, 256, 1) 24 kB (0.011%) : relu_layer_t()
layer[6]  -> (2, 2, 384, 1) 2.1e+04 kB (9.1%) : conv_layer_t(stride=1, kernel_size=3, kernel_count=384, pad=1)
layer[7]  -> (2, 2, 384, 1) 36 kB (0.016%) : relu_layer_t()
layer[8]  -> (2, 2, 384, 1) 3.1e+04 kB (14%) : conv_layer_t(stride=1, kernel_size=3, kernel_count=384, pad=1)
layer[9]  -> (2, 2, 384, 1) 36 kB (0.016%) : relu_layer_t()
layer[10]  -> (2, 2, 256, 1) 2.1e+04 kB (9.1%) : conv_layer_t(stride=1, kernel_size=3, kernel_count=256, pad=1)
layer[11]  -> (2, 2, 256, 1) 24 kB (0.011%) : relu_layer_t()
layer[12]  -> (1, 1, 256, 1) 18 kB (0.0079%) : pool_layer_t(stride=2, filter_size=3, pad=0)
layer[13]  -> (1, 1, 256, 1) 6 kB (0.0026%) : relu_layer_t()
layer[14]  -> (4096, 1, 1, 1) 8.3e+03 kB (3.7%) : fc_layer_t()
layer[15]  -> (4096, 1, 1, 1) 1.3e+05 kB (58%) : fc_layer_t()
layer[16]  -> (10, 1, 1, 1) 3.8e+02 kB (0.17%) : fc_layer_t()
Total 17: 2.3e+05 kB
```

This shows this model has 17 layers. Let's look at layer 0 in particular:

```
IN     (28, 28, 1, 1)
layer[0]  -> (7, 7, 96, 1) 3.2e+02 kB (0.14%) : conv_layer_t(stride=4,
kernel_size=11, kernel_count=96, pad=0)
```

The output above tell us that `layer[0]` outputs a tensor that is 7x7x96x1, its internal parameters consumer 320kB of memory, which is 0.14% of the memory footprint of the whole model. It's a `conv_layer_t` configured with the parameters listed. The configuration of the model for this lab is different. Check your output for the correct values for this lab.
If you want to study the performance of the `activate` and `calc_grads` function in this layer and layers 3, 6, and 10 (the other `conv_layer_t` layers), you use `--test-layer` and `--function` in your `config.env` like this:

```
CMD_LINE_ARGS=--scale 1 --test-layer 0 3 6 10 --function activate
calc_grads
```

If you commit this change and run it with `runlab --run-git-remotely`, you'll get back results in `cnn.csv` that looks like this:

```
dataset|training_inputs_count|param1|param2|param3|param4|full_name                          |function  |layer|layer_type  |reps|WallTime|
-------|---------------------|------|------|------|------|-----------------------------------|----------|-----|------------|----|--------|
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[0]->activate  conv_layer_t   |activate  |0.0  |conv_layer_t|6.0 |1.16    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[3]->activate  conv_layer_t   |activate  |3.0  |conv_layer_t|2.0 |1.23    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[6]->activate  conv_layer_t   |activate  |6.0  |conv_layer_t|5.0 |1.08    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[10]->activate  conv_layer_t  |activate  |10.0 |conv_layer_t|5.0 |1.15    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[0]->calc_grads  conv_layer_t |calc_grads|0.0  |conv_layer_t|2.0 |1.06    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[3]->calc_grads  conv_layer_t |calc_grads|3.0  |conv_layer_t|2.0 |5.45    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[6]->calc_grads  conv_layer_t |calc_grads|6.0  |conv_layer_t|2.0 |2.01    |
mininet|4.0                  |1.0   |1.0   |1.0   |1.0   |layer[10]->calc_grads  conv_layer_t|calc_grads|10.0 |conv_layer_t|2.0 |1.92    |
```

Being selective about what to run has two advantages: First, it's faster than measuring everything, so your autograder submissions will take less time. Second, if you collect a Moneta trace with `mtrace`, it'll be faster too.

Running Specific Tests With `--run-git-remotely`

You can invoke `make` with `runlab` to run just part of the lab, further reducing execution time (and increasing the speed with which you can test things). You can: Run your tests and the benchmark we use for grading:

```
runlab --run-git-remotely
```

Or, equivalently (It's good to refer to the Makefile to check what each target below does.)

```
runlab --run-git-remotely -- make benchmark.csv cnn.csv regressions.out
```

You can just run the tests you've configure with `config.env` with

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

You can also just run tests locally with:

```
runlab -- make regressions.out
```

or

```
make regressions.out
```

You can configure your tests with `config.env`

```
CMD_LINE_ARGS=--function activate --test-layer 0 --scale 5
```

Note the `--scale` option sets (very approximately) the number of seconds the test will run for. 3 is good number – it should give pretty consistent results without taking too long. However, if you want to run lots of tests quickly, you could run at a smaller scale and then confirm with a bigger scale. `--scale 0` will run the function exactly once (which is useful with `mtrace`).

Parameterizing and Iterating Your Tests

`cnn.cpp` has some simple support for running multiple tests in quick succession. It's meant to handle tasks like comparing the performance of multiple implementations of a function or varying tile sizes.
It works like this: The tool defines four global variables (in `parameters.hpp/cpp`):

```
int g_param1_value = 1;
int g_param2_value = 1;
int g_param3_value = 1;
int g_param4_value = 1;
```

And provide four matching sets of command line options (Just two are shown below):

```
--param1-name <name>
--param1 <value> <value> ...
--param2-name <name>
--param2 <value> <value> ...
...
```

For whatever workload you've specified via `--function` and `--test-layer` (see above), it will run it with all possible combination of values for each parameter.
The `--param1-name <name>` name will turn into a column name in the resulting `.csv` file.
For example:

```
./cnn.exe --stats sample.csv --function fix_weights --test-layer 1 2 --
param1-name foo --param1 1 2  --param2-name bar --param2 3 4 --scale 0
pretty-csv sample.csv
```

will yield

```
dataset|training_inputs_count|foo|bar|param3|param4|full_name                          |function   |layer|layer_type  |reps|WallTime|
-------|---------------------|---|---|------|------|-----------------------------------|-----------|-----|------------|----|--------|
mnist  |4.0                  |1.0|3.0|1.0   |1.0   |layer[1]->fix_weights  pool_layer_t|fix_weights|1.0  |pool_layer_t|1.0 |1.19e-06|
mnist  |4.0                  |1.0|4.0|1.0   |1.0   |layer[1]->fix_weights  pool_layer_t|fix_weights|1.0  |pool_layer_t|1.0 |0.0     |
mnist  |4.0                  |2.0|3.0|1.0   |1.0   |layer[1]->fix_weights  pool_layer_t|fix_weights|1.0  |pool_layer_t|1.0 |0.0     |
mnist  |4.0                  |2.0|4.0|1.0   |1.0   |layer[1]->fix_weights  pool_layer_t|fix_weights|1.0  |pool_layer_t|1.0 |0.0     |
```

Testing

READ THIS. IT WILL SAVE YOU LOTS OF GRIEF

The `--param*` mechanism described above makes your life easier because it will let you evaluate multiple implementations of a function in one run. However, this also makes testing more complex.
Since `cnn.exe` can run multiple implementations, `run_tests.exe` (which runs the tests) must be able to do so as well. To support this, it takes the same `--param*` command line arguments as `cnn.exe`.

When you run the tests (`make regressions.out` or `make cnn.csv`), the `Makefile` passes the value of `IMPL_SEL_OPTIONS` to `run_test.exe`. Your can set `IMPL_SEL_OPTIONS` in `config.env`. The `Makefile` also passes `IMPL_SEL_OPTIONS` to `cnn.exe` when you run `make cnn.csv`.
How should use `IMPL_SEL_OPTIONS`? You should put the `--param*` command line arguments that describe the implementation(s) you are testing.
For instance, the starter version of `opt_cnn.hpp` lets you select between two versions of `opt_ft_layer_t::activate`: `opt_fc_layer_t::activate_1` and the original version, `ft_layer_t::activate`.
The

```
#define ACTIVATE_IMPLEMENTATION g_param1_value
```

near the top of `opt_cnn.hpp` and the

```
switch (ACTIVATE_IMPLEMENTATION) {
```

in the body of `opt_fc_layer_t::activate` set things up so `--param1` will control which version of `activate` gets run.
If you wanted to measure the performance of both implementations and run regressions on them, you should set

```
IMPL_SEL_OPTIONS=--param1-name impl --param1 0 1
```

in `config.env`. (The `--param1-name impl` is not strictly necessary, but it will make your `.csv` easier to interpret.)
Then, if you run `make regressions.out` and get

```
...
[==========] 21 tests from 2 test suites ran. (25266 ms total)
[  PASSED  ] 21 tests.
```

Congrats! Your code is correct!

If you've broken something you'll get something like:

```
...
Tests failed for this set of parameters:
impl = 1
param2 = 1
param3 = 1
param4 = 1
```

In this case, you should look in `regressions.out` for more details.
The example above only runs regressions on different implementations. However, changing tile sizes might cause bugs to appear as well. To test for different tile sizes, you can put the `--param*` options that control tile size in `IMPL_SEL_OPTIONS` as well. Be careful, though running the the regressions takes a little while, so running many different permutations make take a looooong time.

Configuring Your Code For Benchmarking

READ THIS TO GET ALL YOUR PERFORMANCE!

When the autograder tests the performance of your implementation of `train_model`, it does not pass `IMPL_SEL_OPTIONS` or `CMD_LINE_ARGS` to `cnn.exe`. This means that your best implementation of each function needs to be the default.
There are several ways you could do this, but I would suggest following the approach taken in the starter code.

You can change this line

```
#define FC_ACTIVATE_IMPLEMENTATION g_param1_value
```

to

```
#define FC_ACTIVATE_IMPLEMENTATION 1
```

and that will make implementation 1 the default, so it will be used by `benchmark.csv`.

So, you can systematically change which function `--param1` controls by following the same pattern with `calc_grads` and `fix_weights` for `fc_layer_t` and other layer types.
You'll need to the same thing for your tile sizes and anything else you else you tune using the `--param*` options.

Run Many Experiments at Once, but not Too Many

Be careful about how many tests you run at once. The runtime of tests (in second) is roughly the product of:

- The number of layers
- The number of functions
- Your `--scale` value
- The number of values provided for each parameter.

Keeping runtimes short is a good idea for a couple reasons. First, it lets you iterate faster. For instance, you may find that you can get useful measurements with `--scale 2`. If that's the case, the do so, you'll be able to try more things more quickly.

Second, if everyone starts running large batches of long-running jobs, wait times will grow rapidly which will be very annoying for everyone. To avoid this `cnn.cpp` implements some limits:

1. You can run up to 144 experiment at once if you pass `--scale 0`.
2. You can run up to 24 experiments at once if `--scale` is in the range 1-3.
3. You can run up to 6 experiments at once if `--scale` is greater than 4. (There's really no need for scale > 4, anyway)

You may already have noticed that `--run-git-remotely` won't let you submit more than once per minute.
We may adjust these limits as we go, depending on the wait times we see on the cluster.

If you have to break up your jobs into smaller batches, you can use `merge-csv` to get everything into one file:
`merge-csv batch1.csv batch2.csv batch3.csv > everything.csv`
Sorry for the hassle, but this is one of the artificial contraints we face in a course setting.

Changes to Moneta

Selecting Traces

There have been some changes in how you start Moneta. You can paste the following into your Jupyter Notebook:

```
from moneta.main import show_trace, select_trace
select_trace()
```

To open up a file browser where you can select a trace.

Or you can specify a path to a trace:

```
from moneta.main import show_trace, select_trace
show_trace("/root/steve/labs/CSE141pp-Lab-Caches/Size-102_0.hdf5")
```

Threads

There is a new "Threads" section in the legend (to the right of main plot). It lets you zoom in on and get statistics for the accesses performed by all the threads running in the program.

Part 1 Starts Here

Experience `-O3`

Let's see how much the compiler is helping us. Set `OPTIMIZE=-O3` in `config.env`, and measure performance for our benchmark:

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

P1 (1pt): Provide the runtime for `train_model` reported in `benchmark.csv`.

```
-O3 execution time:
```

The set `OPTIMIZE=-O0` and run it again.

P1 (1pt): Provide the runtime for `train_model` in `benchmark.csv`.

```
-O0 execution time:

Speedup from -O3:
```

Get Yourself a Map

Next, we analyze the performance of the existing code and figure out which functions are taking majority of the execution time in this new CNN architecture of ours.

To generate benchmark results, first enable `gprof` by uncomment the line `GROF=yes` and set `OPTIMIZE=-O3`. Commit, push, and run `make benchmark.csv` again with `--run-git-remotely`.

Examine the output in `benchmark.gprof` and answer the following:

P3 (3pt): List the functions (as listed in `benchmark.gprof`) that combined to account for 99% of the total execution time. Report the percent proportions as well.

Note: Remember to turn of `gprof` when you dont need it. It adds overhead.

Tile `fc_layer_t::activate`

Add a new implemententation of `fc_layer_t::activate` that tiles all three loops based on the iteration space analysis I did during the lecture for the lab.
Compare the performance of the resulting code to the original code in `fc_layer_t.hpp` (i.e, `fc_layer_t::activate`) on layer 14 of the model. You should get a speedup of 9-10x.
Use the `--param*` command line options to do this in one run.

P3 (3pt) Write the `runlab` command and the values of `CMD_LINE_ARGS` and `IMPL_SEL_ARGS` used to compare your optimization with the original code

```
runlab command:

CMD_LINE_ARGS:

IMPL_SEL_ARGS:
```

P4 (1pt) Note down the both execution times, and calculate the speedup

```
Original execution time:

Optimised execution time:

Speedup:
```

P5 (4pt) Use `--param*` to explore the range of tiling sizes for `activate()`. Draw a graph that shows the impact on execution time of tiling `I` and `N` at different sizes. Make sure it is clearly labeled legible. You are plotting execution time against two variables, so you will need to account for that in the graph.

```
Your graph here
```

P6 (2pt) Based on your data, what are the optimal tile sizes? How much speedup do the optimal sizes provide relative to the performance you measured for P4?

```
Optimal I_TILE_SIZE:

Optimal B_TILE_SIZE:

Optimal N_TILE_SIZE:

Speedup of Optimal over P4:
```

Go forth and optimize!

This portion of the project is due at the end of the final project.

With `gprof` as your guide, optimize the execution time of `train_model` as much as possible. You can use compiler flags, tiling, multi-threading (see Part 2), vectorization (see Part 3), and any manual modification of the code you can implement in `opt_cnn.hpp`.
There are a few caveats:

1. You can't modify `cnn.cpp`
2. Your code must pass all the regression tests.
3. You can, and should, use your code from the previous labs if you think they had good performance.

The target speedup is 7x on `train_model`. You'll need to combine multiple techniques (tiling, threading, etc.) to achieve the target.
A few things to try/think about:

1. Let `gprof` be your guide.
2. You can pass whatever compiler flags you'd like to `OPTIMIZE`.
3. Consider loop order (to maximize spatial locality).
4. Consider tiling (to maximize temporal locality).
5. Consider loop unrolling.
6. There are many potential targets. Be systematic.
7. Make use of the `--param*` mechanism. Follow the model in the starter code for controlling things from the command line. Time invested in being able to run experiments efficiently will pay off over the course of the project.

Grading on the Optimization Portion

READ ALL OF THIS. IT WILL SAVE YOU SOME STRESS

Your grade for the optimization port of the final project is just the fraction of the 7x speedup you achieve *at the end of the project*.

The execution time for the baseline implementation of `train_model` as measured by `benchmark.csv` is 22.7 seconds. So, if you're final implementation runs in 10s, your speedup is 2.27 and your grade on the optimization portion of the projuct will be 2.27/7 = 32%.
To help you gauge your progress, here are some target speedups for the two checkpoints:

- Checkpoint 1: If you are at about 1.6x at the first check point, you are doing fine (assumes no threads or vectorization).
- Checkpoint 2: If you are at between 4-6x at the second check point, you are doing fine (assumes no vectorization).

In both cases "doing fine" means that you're likely to get a good grade if you continue to work diligently on the project. Your grade is solely determined by the speedup of your final submission.

YOU ARE NOT DONE. THERE IS ANOTHER PART (`README2.pdf`)
