High Performance Computing
Homework #5: Phase 2
Due: Wed Oct 27 2021 before 11:59 PM
Email-based help Cutoff: 5:00 PM on Tue, Oct 26 2021

> **!** The runtime data and results in this report are meaningful only if your implementation is functionally correct and produce similar outputs as the reference run.

Name:

Experimental Platform

The experiments documented in this report were conducted on the following platform:

Component	***Details***
CPU Model	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
CPU/Core Speed	2.40GHz
Operating system used	Unix
Interconnect type & speed (if applicable)	Not applicable
Was machine dedicated to task (`yes`/`no`)	Yes (via a `slurm` job)
Name and version of C++ compiler (if used)	
Name and version of Java compiler (if used)	None
Name and version of other non-standard software tools & components (if used)	

Runtime data for the reference performance

In the table below, record the reference runtime characteristics. This is the data for your enhanced version from Phase #1:

Rep	User time (sec)	Elapsed time (sec)	Peak memory (KB)
1	35.77	36.59	3516
2	34.26	34.9	3516
3	33.62	34.26	3516
4	34.78	35.43	3516
5	35.69	36.41	3516

Perf report data for the reference implementation

In the space below, copy-paste the `perf` profile data that you used to identify the aspect/method to reimplement to improve performance:

Attached in the submission, I am not sure why but I cannot open it anymore

Description of performance improvement

Briefly describe the performance improvement you are implementing. Your description should document:

- Why you chose the specific aspect/feature to improve (obviously it should be supported by your `perf` data)
- What is the best-case improvement that you anticipate – for example, if you optimize a feature that takes 25% of runtime, then the best case would be a 25% reduction in runtime.
- Briefly describe what/how you plan to change the implementation

I improved performance by caching images to the unorder map, it improved the performance I decided to save the images into unordered map.

```
    const std::string testImgs  = (argc > 5 ? argv[5] : "TestingSetList.txt");
    // Create the neural netowrk
    NeuralNet net({784, 30, 10});
    // Train it in at most 30 epochs.
    imagepair training_data = loadimages(argv[1], "TrainingSetList.txt", c);
    imagepair test_data = loadimages(argv[1], "TestingSetList.txt", c);
    for (int i = 0; (i < epochs); i++) {
        std::cout << "-- Epoch #" << i << " --\n";
        std::cout << "Training with " << c << " images...\n";
        const auto startTime = std::chrono::high_resolution_clock::now();
        for (int i = 0; i < c; i++)
            net.learn(training_data.first.at(i), training_data.second.at(i));
        train(net, argv[1], training_data, c, trainImgs);
        assess(net, test_data);
        const auto endTime = std::chrono::high_resolution_clock::now();
        // Compute the timeelapsed for this epoch
        using namespace std::literals;
        std::cout << "Elapsed time = " << ((endTime - startTime) / 1ms)
                  << " milliseconds.\n";
    }
    return 0;
}
```

```
imagepair loadimages(std::string path, const std::string& imgFileList,
        int limit = 100) {
    std::ifstream fileList2(imgFileList);
    std::vector<std::string> fileNames;
    int count = 0;
    // Load the data from the given image file list.
    for (std::string imgName; std::getline(fileList2, imgName) &&
            count < limit; count++) {
        fileNames.push_back(imgName);
    }
    // Randomly shuffle the list of file names so that we use a random
    // subset of PGM files for training.
    std::default_random_engine rg;
    std::shuffle(fileNames.begin(), fileNames.end(),
                 std::default_random_engine());

    image_map trainimgs;
    image_map trainlabels;
    // Check how many of the images are correctly classified by the
    // given given neural network.
    int i = 0;
    for (std::string imgName : fileNames) {
        // if (i%100 == 0)
        //     std::cout << imgName << '\n';
        Matrix img = loadPGM(path + "/" + imgName);
        Matrix exp = getExpectedDigitOutput(imgName);
        trainimgs.insert(std::pair(i, img));
        trainlabels.insert(std::pair(i, exp));
        i++;
    }
    return imagepair(trainimgs, trainlabels);
}
```

That saved time which was spent on reading images files each epoch. Unfortunately, here we run into main limitation:

- If we agree to train on the same images each epochs, performance improves a lot (user time decreases until 25.7 sec and elapsed time is around 35 sec)

What is interesting, using this approach does not decrease the accuracy and (similar to the one obtained it is around 31%.

```
-- Epoch #0 --
Training with 5000 images...
Correct classification: 997 [0.19944% ]
Elapsed time = 2454 milliseconds.
-- Epoch #1 --
Training with 5000 images...
Correct classification: 1539 [0.307862% ]
Elapsed time = 2433 milliseconds.
-- Epoch #2 --
Training with 5000 images...
Correct classification: 1580 [0.316063% ]
Elapsed time = 2454 milliseconds.
-- Epoch #3 --
Training with 5000 images...
Correct classification: 1595 [0.319064% ]
Elapsed time = 2387 milliseconds.
```

Unfortunately, if we decide to create that map each epoch the user time would be 45 second and elapsed over 108 sec. That is caused by the fact that we are creating a new object like that each time.

Thanks to creating **loadimages** function, I was able to stop using **train** function and substitute it by 2 line for loop which implement inside the main function (methods inlining).

Runtime statistics from performance improvement

Use the supplied SLURM script to collect runtime statistics for your enhanced implementation.

Rep	User time (sec)	Elapsed time (sec)	Peak memory (KB)
1	29.40	36.41	656572
2	28.06	39.06	656572
3	26.77	33.18	656572
4	26.41	32.70	656572
5	26.57	33.67	656572

Comparative runtime analysis

Compare the runtimes (*i.e.*, before and after your changes) by fill-in the Runtime Comparison Template and copy-paste the full sheet in the space below:

Change tile for cases and the type of data you are entering		
Replicate#	**Reference**	**Improved**
1	29.4	35.77
2	28.06	34.26
3	26.77	33.62
4	26.41	34.78
5	26.57	35.69
Average:	27.442	34.824
SD:	1.274036891	0.9239209923
95% CI Range:	1.58192574	1.147199433
Stats:	**27.442 ± 1.58**	**34.824 ± 1.15**
T-Test (H_0: μ1=μ2)	0.00001164774596	

Change tile for cases and the type of data you are entering		
Replicate#	**Reference**	**Improved**
1	36.41	36.59
2	39.06	34.9
3	33.18	34.26
4	32.7	35.43
5	33.67	36.41
Average:	35.004	35.518
SD:	2.685280246	0.9895807193
95% CI Range:	3.334215807	1.228726752
Stats:	**35.004 ± 3.33**	**35.518 ± 1.23**
T-Test (H_0: μ1=μ2)	0.70435008	

Inferences & Discussions

Now, using the data from the runtime statistics discuss (at least 5-to-6 sentences) the change in runtime characteristics (both time and memory) due to your changes. Compare and contrast key aspects/changes to the implementation. Include any additional inferences as to why one version performs better than the other.

While I managed to significantly decrease user time, peak memory significantly increased, as well as slight increase in elapsed time in some of the runs (and decrease in another ones 😊. That was an expected thing to see (at the end we save multiple images and keeping them in the memory). Since unorder map has constant iteration time complexity (O(1)) independent of number of samples. Additionally map is kept in the memory and allows on the faster access. Further performance can be definitely improved by using OpenMP to parallelize for loops. Analyzing Ttest we can also observed that a large difference exists between the two sample sets.