# High Performance Computing
# Homework #5: Phase 2
# Due: Wed Oct 26 2022 before 11:59 PM
# Email-based help Cutoff: 5:00 PM on Tue, Oct 25 2022

> **!** 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:** Josh Crail

## *Experimental Platform*

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

| ***Component*** | ***Details*** |
|---|---|
| CPU Model | pitzer-login02.hpc.osc.edu |
| CPU/Core Speed | 2.40 GHz |
| Operating system used | Linux |
| 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 | 20.98 | 21.61 | 649716 |
| 2 | 20.86 | 21.43 | 649720 |
| 3 | 21.13 | 21.70 | 649716 |
| 4 | 21.00 | 21.57 | 649720 |
| 5 | 21.02 | 21.61 | 649716 |

## *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:

```
-  91.24%     0.00%  homework5  homework5           [.] main
   - main
      - 55.27% train
         - train
            + 50.49% NeuralNet::learn
            + 4.28% loadPGM
            + 0.51% std::operator+<char, std::char_traits<char>, std::allocator<char> > (inlined)
      - 35.97% assess
         - 28.62% NeuralNet::classify
            + 17.18% Matrix::dot
            + 6.74% Matrix::Matrix (inlined)
            + 2.80% Matrix::operator= (inlined)
            + 0.96% Matrix::~Matrix (inlined)
              0.94% Matrix::apply<double (double)> (inlined)
         + 6.39% loadPGM
```

## *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 am choosing to improve the classify method. This is because when looking at the main in the assess method we can see that classification is 28.62% of the 35.97%. In the classify method we can see that the Matrix::dot takes a lot of the percentage so I will try to better optimize (everything else is not really optimizable). I am hoping to get the performance of dot to be 25% better meaning that it should take 25% less time for the program to use the dot for classify. So we should see about a two second performance increase (given 25 seconds * (.28 (initial percentage * .25 (performance increase) ≈ 2 seconds.

## *Source code changes for performance improvement*

Copy-paste parts of the program that you actually modified to improve performance:

**Changes to Matrix.h/.cpp (if any)**

```
 static Val sigmoid(const Val val) {
   return 1. / (1. + std::exp(-val));
 }

 // Definition of sigmoid in the Matrix.h

void dot2(const Matrix& rhs, const Matrix other, Matrix& result) const;
 // method for the new dot method

void Matrix::dot2(const Matrix& rhs, const Matrix other, Matrix& result) const
{
    // Ensure the dimensions are similar.
    // std::cout << width() << ", height = " << rhs.height() << std::endl;
    assert(this->width() == rhs.height());
    // assert(front().size() == rhs.size());
    // Setup the result matrix
    const auto mWidth = rhs.front().size(), width = front().size();
    result = Matrix(size(), mWidth);
    // Do the actual matrix multiplication
    for (size_t row = 0; (row < size()); row++) {
        for (size_t col = 0; (col < mWidth); col++) {
            Val sum = 0;
            for (size_t i = 0; (i < width); i++) {
                sum += (*this)[row][i] * rhs[i][col];
            }
            // Store the result in an appropriate entry
            // and do the addtion right here
            // sigmoid is this
            // 1. / (1. + std::exp(-val))
            result[row][col] = (sum + other[row][col]);
        }
    }
    // Return the computed result and apply sigmoid
    result.apply(sigmoid);
    // return result;
}
// This is the new dot method where it does addition and sigmoid that was done
in classify in the method to reduce the overhead and increase the performance
```

**Changes to NeuralNet.h/.cpp (if any)**

```
 void classify(const Matrix& inputs, Matrix& result) const;
// New definition for the classify method


// The method to classify/recognize a given input.
void
NeuralNet::classify(const Matrix& inputs, Matrix& result) const {
    result = inputs;
    Matrix finalResult = inputs;
    for (size_t lyr = 0; (lyr < weights.size()); lyr++) {
        // CHANGES
        weights[lyr].dot2(result, biases[lyr], finalResult);
        result = finalResult;
    }
    result = finalResult;
}
// Changed the classify so that it calls my new dot method and
// that it is a void now so that there is no need to create a result matrix
```

**Changes to main.cpp (if any)**

```
    for (std::string imgName; std::getline(fileList2, imgName); totCount++) {
        loadPGM(path + "/" + imgName, img);
        exp = getExpectedDigitOutput(imgName);
        // Have our network classify the image.
        net.classify(img, res);
        assert(res.width() == 1);
        assert(res.height() == 10);        // CHANGES
        if (maxElemIndex(exp.transpose()[0]) ==
            maxElemIndex(res.transpose()[0])) {
                passCount++;
            }    }

// Changes this is the assess method so that we are not saving the int return
// from the method instead just comparing it directly
// not a major change but it is better than assigning an int for no reason
// changed the net.classify to be a void and pass the result by ref
```

## *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 | 19.13 | 19.72 | 649760 |
| 2 | 19.20 | 19.85 | 649756 |
| 3 | 19.14 | 19.75 | 649760 |
| 4 | 19.12 | 19.66 | 649760 |
| 5 | 19.09 | 19.68 | 649756 |

## *Perf report data for the revised implementation*

In the space below, copy-paste the `perf` profile data that highlights the effectiveness of your reimplementation to improve performance:

```
- 81.10%  0.00%  homework5  homework5        [.] main
  - main
    - 41.34% train
      - train
        + 39.66% NeuralNet::learn
        + 1.68% loadPGM
    - 39.77% assess
      - 33.04% NeuralNet::classify
        + 18.96% Matrix::dot2
        + 7.33% Matrix::Matrix (inlined)
        + 6.75% Matrix::operator= (inlined)
      + 6.31% loadPGM
```

## *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:

| Replicate# | Reference | Improved |
|---|---|---|
| 1 | 21.61 | 19.72 |
| 2 | 21.42 | 19.85 |
| 3 | 21.7 | 19.75 |
| 4 | 21.57 | 19.66 |
| 5 | 21.61 | 19.68 |

| | | |
|---|---|---|
| **Average:** | 21.582 | 19.732 |
| **SD:** | 0.1023230179 | 0.07463243263 |
| **95% CI Range:** | 0.1270508076 | 0.09266840469 |
| **Stats:** | **21.582 ± 0.13** | **19.732 ± 0.09** |
| **T-Test ($H_0$: μ1=μ2)** | 0.00000000338046493 | |

## *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.

It seems like the peak memory has increased by about (30~40 KB) not really important when you compare it to the total KB that we are already at. We can see that the main is now 81.10% vs the 91.24% it was at before so we can see that the dot2 method did make a difference. The percentage of dot2 is higher but there was a reduction in time and the overall percentage is down. The key change here is that there is no longer a call to dot then the + operator, and the apply(sigmoid) these are all condensed down a single method call to dot2. With all of those calls reduced down to a single method call it makes sense that the time is reduced, the time reduction is around what I guessed it to be, but at the same time I was hoping this implementation would give more speed.