DUE DATE: **Phase #1: Due Thu Oct 17 2024 before 11:59 PM**

High Performance Computing
Homework #4: Phase 1
Due: Thu Oct 17 2024 before 11:59 PM

!	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: Nicholas McCarty

Experimental Platform

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

Component	***Details***
CPU Model	Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
CPU/Core Speed	2400mhz
Operating system used	Linux 64x
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)	g++=17
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 of the starter code:

Rep	**User time (sec)**	**Elapsed time (sec)**	**Peak memory (KB)**
1	36.06	36.85	3176
2	35.62	36.46	3175

3	35.49	36.19	3180
4	35.74	36.59	3176
5	35.74	36.59	3176

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:

```
---_start
          __libc_start_main
          main
          |
          |--43.78%--assess
          |        |
          |        |--27.57%--loadPGM
          |        |        |
          |        |        |--17.18%--std::istream::operator>> (inlined)
          |        |        |        std::istream::_M_extract<double>
          |        |        |        |
          |        |        |        |--15.52%--std::num_get<char, std::istreambuf_iterator<char,
std::char_traits<char> > >::get (inlined)
          |        |        |        |        |
          |        |        |        |         --14.96%--std::num_get<char,
std::istreambuf_iterator<char, std::char_traits<char> > >::do_get
          |        |        |        |                  |
```

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

DUE DATE: **Phase #1: Due Wed Oct 20 2021 before 11:59 PM**

I am going to attempt to change the LOADPGM method so we can get better runtime results. It has an atrocious 27% perf percentage and I want to get this to around 20%. My plan of attack for optimizing the PGM method goes as follows: I'm going to use buffered input by reading pixel values into a std::vector<int> and normalizing them afterward, which will reduce the overhead from multiple stream operations. Additionally, I will pre-allocate the Matrix with its full size upfront to avoid costly dynamic resizing during the matrix's population.

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)

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

Changes to main.cpp (if any)

```
Matrix loadPGM(const std::string& path) {
   std::ifstream file(path, std::ios::in | std::ios::binary);
   if (!file.is_open()) {
       throw std::runtime_error("Unable to read " + path);
   }


   // Read header
   std::string hdr;
   int width, height;
   Val maxVal;
   file >> hdr >> width >> height >> maxVal;

   if (hdr != "P2") {
       throw std::runtime_error("Only P2 PGM format is
supported");
   }

   // Ignore whitespace between header and data
   file.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');


   // Pre-allocate the matrix with known size
   Matrix img(width * height, 1);

   // Read all the pixel values in one go
   std::vector<int> buffer(width * height);
   for (int& pixel : buffer) {
       file >> pixel;
   }


   // Normalize the pixel values and store them in the matrix
   Val invMaxVal = 1.0 / maxVal;
   for (int i = 0; i < width * height; ++i) {
       img[i][0] = buffer[i] * invMaxVal;
   }


   return img;
}
```

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	28.56	29.33	3168
2	28.28	28.98	3168
3	28.44	29.19	3168
4	28.23	28.36	3172
5	28.20	28.92	3168

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:

```
|        |--9.54%--loadPGM
|        |        |
|        |        |--5.75%--std::istream::operator>>
|        |        |        |
|        |        |        |--4.17%--std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::get (inlined)
|        |        |        |        |
|        |        |        |         --3.70%--std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
|        |        |        |
|        |        |         --0.95%--std::istream::sentry::sentry
|        |        |
|        |        |--2.21%--Matrix::Matrix
|        |        |          std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<double> > > >::vector (inlined)
|        |        |          std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<double> > > >::_M_fill_initialize (inlined)
|        |        |          std::__uninitialized_fill_n_a<std::vector<double, std::allocator<double> >*, unsigned long, std::vector<double, std::allocator<double> >, std::vector<double, std::allocator<double> > > (inlined)
|        |        |          std::uninitialized_fill_n<std::vector<double, std::allocator<double> >*, unsigned long, std::vector<double, std::allocator<double> > > (inlined)
```

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:

A	B	C
Change tile for cases and the type of data you are entering		
Replicate#	**Reference**	**Improved**
1	36.85	29.33
2	36.46	28.98
3	36.19	29.19
4	36.59	28.36
5	36.59	28.92
Average:	36.536	28.956
SD:	0.2397498697	0.3713892836
95% CI Range:	0.2976887817	0.4611407028
Stats:	**36.536 ± 0.3**	**28.956 ± 0.46**
T-Test (H_0: μ1=μ2)	0.000000003058	

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.

From the clear outcome analyzed from my program enhancement, the optimized embodiment has a specific characteristic, in that it remarkably lessens both run-time and memory allocation comparing to the initial case. The most immediate development is alter the loadPGM approach which turned out reducing its percent of total runtime from 27% to 9.54%. The above accomplishment was due to the optimal usage of an input buffer and a pre-allocated matrix size. The cut in the memory overhead resulting from the elimination of the unnecessary dynamic allocations is also clear. In general, the improved version not only executes quicker but also eats resources better, which definitely makes it a scalable solution for more extensive datasets.