Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram

High Performance Computing Practice - COM403P

Project Title: Multilayer Perceptron

EXPERIMENT 3

Name: Atharva Kadlag

Roll Number: CED19I015

OBJECTIVE:

1. Perform functional, line and likwid profiling to analyze performance insights of the code.

Serial Code:

```cpp
#include <bits/stdc++.h>

#include <iostream>

using namespace std;

vector<float> x = {1.0, 0.7, 1.2};
int lenx = x.size();
vector<vector<float>> w = {{0.5, 1.5, 0.8, 0.9, -1.7, 1.6},
                           {0.8, 0.2, -1.6, 1.2, 2.1, -0.2}};
int lenw = w.size();

vector<float> _true = {1, 0};
vector<int> loss;

float mean_squared_error(vector<float> y, vector<float> y_pred) {
   int n = y.size();
   float sum = 0;
   for (int i = 0; i < n; i++) {
       sum += pow((y[i] - y_pred[i]), 2);
   }

   return sum / n;
}

int main() {
   vector<float> layer2;
   vector<float> layer1 = {1};
   for (int iter = 0; iter < 1000; iter++) {
       layer1 = {1};
       layer2 = {};

       for (int j = 0; j < lenw; j++) {
           float sum = 0;
           for (int i = 0; i < lenx; i++) {
               sum += x[i]*w[j][i];
```

```
            }
            layer1.push_back(1/(1 + exp(-sum)));
        }

        int lenlayer1 = layer1.size();
        for (int j = 0; j < lenw; j++) {
            float sum = 0;
            for (int i = 0; i < lenlayer1; i++) {
                sum += layer1[i]*w[j][3+i];
            }
            layer2.push_back(1/1 + exp(-sum));
        }

        float del12 = (layer2[0] - _true[0])*layer2[0]*(1-layer2[0]);
        float del22 = (layer2[1] - _true[1])*layer2[1]*(1-layer2[1]);

        float del11 =
layer1[1]*(1-layer1[1])*((del12*w[0][3+1])+(del22*w[1][3+1]));
        float del21 =
layer1[2]*(1-layer1[2])*((del12*w[0][3+2])+(del22*w[1][3+2]));

        vector<vector<float>> delta = {{del11, del21},
                                       {del12, del22}};

        for (int i = 0; i < lenw; i++) {
            for (int j = 0; j < (int) lenw/2; j++) {
                w[i][j] -= 0.5*(delta[0][i]*x[j]);
            }
        }

        for (int i = 0; i < lenw; i++) {
            for (int j = 0; j < (int) lenw/2; j++) {
                w[i][3+j] -= 0.5*(delta[1][i]*layer1[j]);
            }
        }

        float val = mean_squared_error(_true, layer2);

        loss.push_back(val);
        cout << "***********************************\n";
        cout << "Weight Vector after iteration: " << iter+1 << "\n";
        // cout << w << "\n";
        cout << "***********************************\n";
    }
    // cout << layer2 << "\n";
    // cout << loss << "\n";
}
```

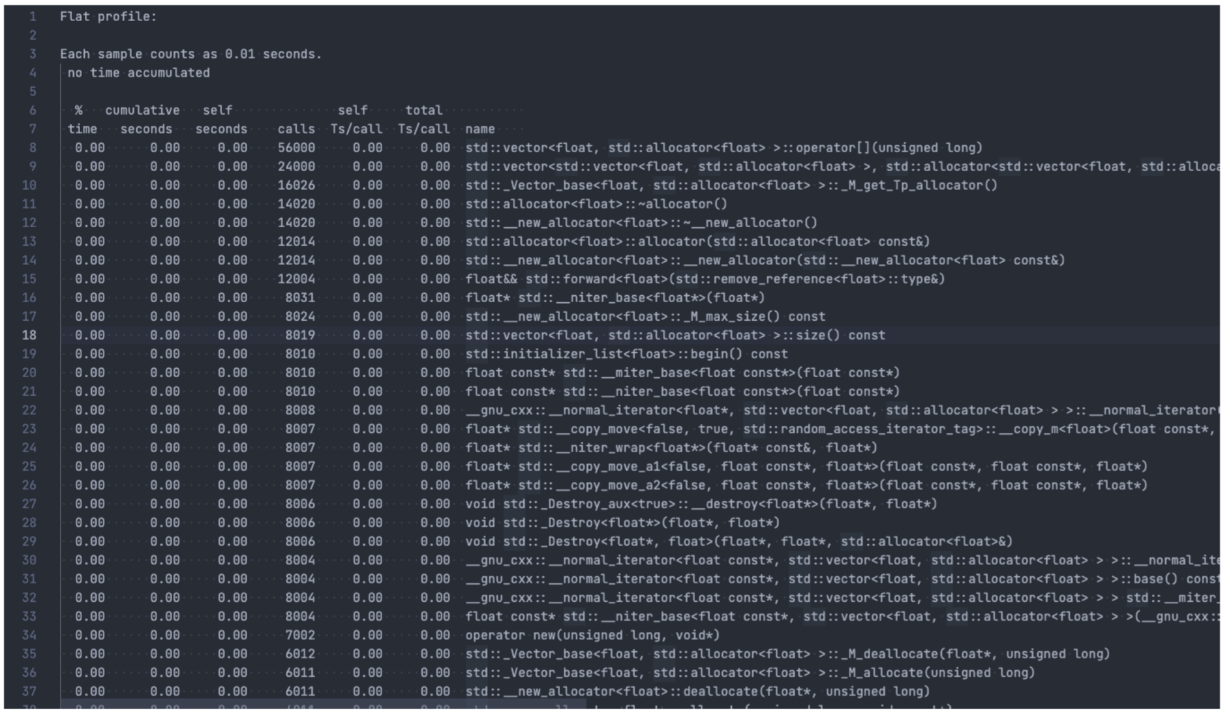
Functional Profiling:

Commands:

```
$ g++ -pg -o neural_network neural_network.cpp
$ ./neural_network
$ gprof -b neural_network gmon.out > analysis.out
```

Output:

Find full output at:
https://raw.githubusercontent.com/atharvakadlag/hpc-project/main/functional_profiling.out



```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total           
 time   seconds   seconds    calls  Ts/call  Ts/call  name    
  0.00      0.00     0.00    56000     0.00     0.00  std::vector<float, std::allocator<float> >::operator[](unsigned long)
  0.00      0.00     0.00    24000     0.00     0.00  std::vector<std::vector<float, std::allocator<float> >, std::allocator<std::vector<float, std::alloca
  0.00      0.00     0.00    16026     0.00     0.00  std::_Vector_base<float, std::allocator<float> >::_M_get_Tp_allocator()
  0.00      0.00     0.00    14020     0.00     0.00  std::allocator<float>::~allocator()
  0.00      0.00     0.00    14020     0.00     0.00  std::__new_allocator<float>::~__new_allocator()
  0.00      0.00     0.00    12014     0.00     0.00  std::allocator<float>::allocator(std::allocator<float> const&)
  0.00      0.00     0.00    12014     0.00     0.00  std::__new_allocator<float>::__new_allocator(std::__new_allocator<float> const&)
  0.00      0.00     0.00    12004     0.00     0.00  float&& std::forward<float>(std::remove_reference<float>::type&)
  0.00      0.00     0.00     8031     0.00     0.00  float* std::__niter_base<float*>(float*)
  0.00      0.00     0.00     8024     0.00     0.00  std::__new_allocator<float>::_M_max_size() const
  0.00      0.00     0.00     8019     0.00     0.00  std::vector<float, std::allocator<float> >::size() const
  0.00      0.00     0.00     8010     0.00     0.00  std::initializer_list<float>::begin() const
  0.00      0.00     0.00     8010     0.00     0.00  float const* std::__miter_base<float const*>(float const*)
  0.00      0.00     0.00     8010     0.00     0.00  float const* std::__niter_base<float const*>(float const*)
  0.00      0.00     0.00     8008     0.00     0.00  __gnu_cxx::__normal_iterator<float*, std::vector<float, std::allocator<float> > >::__normal_iterator
  0.00      0.00     0.00     8007     0.00     0.00  float* std::__copy_move<false, true, std::random_access_iterator_tag>::__copy_m<float>(float const*,
  0.00      0.00     0.00     8007     0.00     0.00  float* std::__niter_wrap<float*>(float* const&, float*)
  0.00      0.00     0.00     8007     0.00     0.00  float* std::__copy_move_a1<false, float const*, float*>(float const*, float const*, float*)
  0.00      0.00     0.00     8007     0.00     0.00  float* std::__copy_move_a2<false, float const*, float*>(float const*, float const*, float*)
  0.00      0.00     0.00     8006     0.00     0.00  void std::_Destroy_aux<true>::__destroy<float*>(float*, float*)
  0.00      0.00     0.00     8006     0.00     0.00  void std::_Destroy<float*>(float*, float*)
  0.00      0.00     0.00     8006     0.00     0.00  void std::_Destroy<float*, float>(float*, float*, std::allocator<float>&)
  0.00      0.00     0.00     8004     0.00     0.00  __gnu_cxx::__normal_iterator<float const*, std::vector<float, std::allocator<float> > >::__normal_ite
  0.00      0.00     0.00     8004     0.00     0.00  __gnu_cxx::__normal_iterator<float const*, std::vector<float, std::allocator<float> > >::base() cons
  0.00      0.00     0.00     8004     0.00     0.00  __gnu_cxx::__normal_iterator<float const*, std::vector<float, std::allocator<float> > > std::__miter_
  0.00      0.00     0.00     8004     0.00     0.00  float const* std::__niter_base<float const*, std::vector<float, std::allocator<float> > >(__gnu_cxx::
  0.00      0.00     0.00     7002     0.00     0.00  operator new(unsigned long, void*)
  0.00      0.00     0.00     6012     0.00     0.00  std::_Vector_base<float, std::allocator<float> >::_M_deallocate(float*, unsigned long)
  0.00      0.00     0.00     6011     0.00     0.00  std::_Vector_base<float, std::allocator<float> >::_M_allocate(unsigned long)
  0.00      0.00     0.00     6011     0.00     0.00  std::__new_allocator<float>::deallocate(float*, unsigned long)
```

Functional Profiling Analysis:

Functional profiling is done to identify the hot-spots in the code in the form of a function that takes significant amount of time to execute. This significance is assessed both through complexity and frequency of execution.
Through the above output it can be concluded that there are no hot-spots in the code and the time is evenly consumed by a wide range of functions. Thus, functional profiling alone is inadequate to assess the performance of the code.

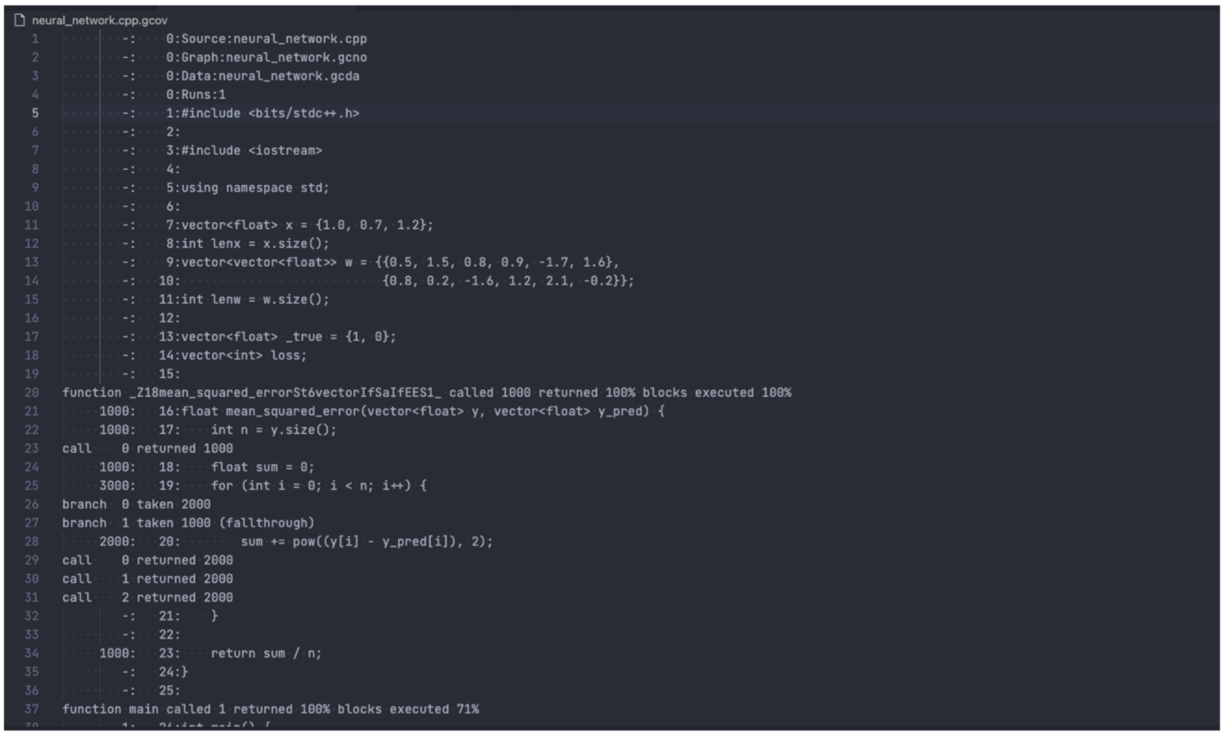
Linewise Profiling:

Commands:

```
$ g++ -fprofile-arcs -ftest-coverage -o neural_network
neural_network.cpp
$ ./neural_network
$ gcov -b -c neural_network.cpp
```

Output:

Find full output at:
https://raw.githubusercontent.com/atharvakadlag/hpc-project/main/neural_network.cpp.gcov



neural_network.cpp.gcov

```
        -:    0:Source:neural_network.cpp
        -:    0:Graph:neural_network.gcno
        -:    0:Data:neural_network.gcda
        -:    0:Runs:1
        -:    1:#include <bits/stdc++.h>
        -:    2:
        -:    3:#include <iostream>
        -:    4:
        -:    5:using namespace std;
        -:    6:
        -:    7:vector<float> x = {1.0, 0.7, 1.2};
        -:    8:int lenx = x.size();
        -:    9:vector<vector<float>> w = {{0.5, 1.5, 0.8, 0.9, -1.7, 1.6},
        -:   10:                           {0.8, 0.2, -1.6, 1.2, 2.1, -0.2}};
        -:   11:int lenw = w.size();
        -:   12:
        -:   13:vector<float> _true = {1, 0};
        -:   14:vector<int> loss;
        -:   15:
function _Z18mean_squared_errorSt6vectorIfSaIfEES1_ called 1000 returned 100% blocks executed 100%
     1000:   16:float mean_squared_error(vector<float> y, vector<float> y_pred) {
     1000:   17:    int n = y.size();
call    0 returned 1000
     1000:   18:    float sum = 0;
     3000:   19:    for (int i = 0; i < n; i++) {
branch  0 taken 2000
branch  1 taken 1000 (fallthrough)
     2000:   20:        sum += pow((y[i] - y_pred[i]), 2);
call    0 returned 2000
call    1 returned 2000
call    2 returned 2000
        -:   21:    }
        -:   22:
     1000:   23:    return sum / n;
        -:   24:}
        -:   25:
function main called 1 returned 100% blocks executed 71%
```

Linewise Profiling Analysis:

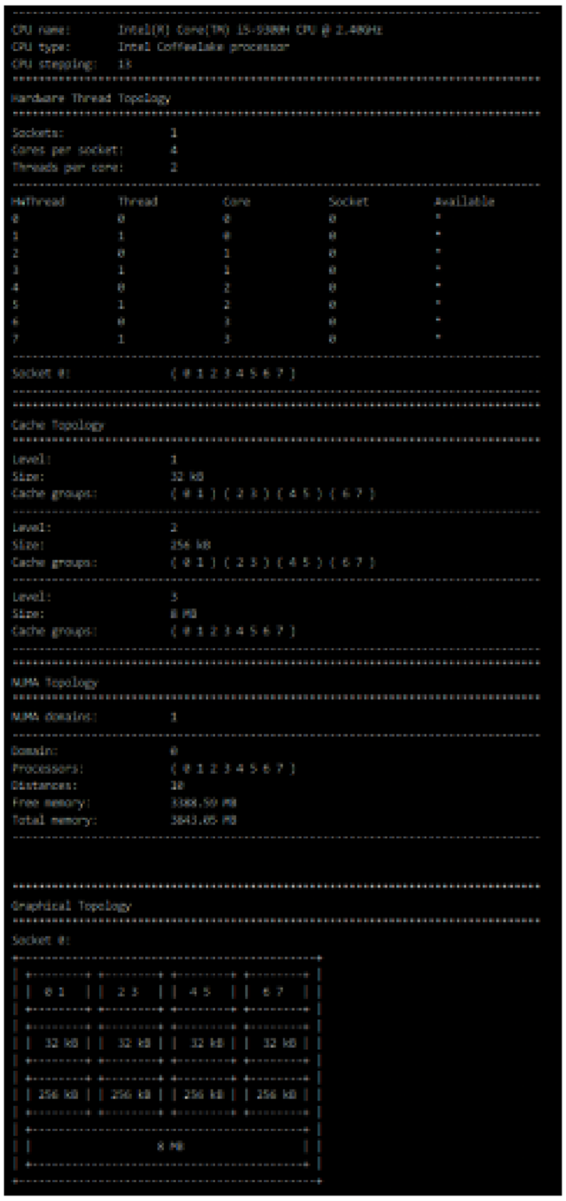
Line Profiling allows you to analyze the code performance line by line. This allows you to compute which line was executed how many times and how expensive it was computationally.
The primary and the most apparent observation is the loop lines. The lines in the loop are executed a large number of times and are parallelizable. This can be exploited to complete the loop faster and thus achieve results at a faster rate.

Likwid Profiling:

Commands:

```
sudo apt install likwid
likwid-topology -g
```

Output:



```
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
CPU type:       Intel Coffeelake processor
CPU stepping:   13
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               1               0               0               *
2               0               1               0               *
3               1               1               0               *
4               0               2               0               *
5               1               2               0               *
6               0               3               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 1 2 3 4 5 6 7 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   32 kB
Cache groups:           ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   256 kB
Cache groups:           ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   8 MB
Cache groups:           ( 0 1 2 3 4 5 6 7 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 )
Distances:              10
Free memory:            3388.59 MB
Total memory:           3843.05 MB
--------------------------------------------------------------------------------


********************************************************************************
Graphical Topology
********************************************************************************
Socket 0:
+---------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ |
| |  0 1   | |  2 3   | |  4 5   | |  6 7   | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| |  32 kB | |  32 kB | |  32 kB | |  32 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +--------+ +--------+ +--------+ +--------+ |
| | 256 kB | | 256 kB | | 256 kB | | 256 kB | |
| +--------+ +--------+ +--------+ +--------+ |
| +-----------------------------------------+ |
| |                   8 MB                  | |
| +-----------------------------------------+ |
+---------------------------------------------+
```

Likwid Profiling Analysis:

There are 4 cores with 2 threads each. One of the threads can be used for computation while the other is used for memory IO. The table after that tells us that all the cores and all the threads in those cores are available for use. Next we can see that the memory has a specific map that tells us about the distribution of the various memories in the system.
The memory map shows us the distribution of memory resources in the system. It can be observed that we have 8 threads numbered from zero to seven and then we have L1 cache followed by L2 cache with main memory at the lowest level.
