CSE 625 Term Project Report

<Project Title>

Caleb Klenda

12-07-2022

# Project statement and objective

The purpose of this project is to compare the effectiveness of different parallelized computing techniques. A portion of the project is dedicated to doing this using OpenMP, an open source library for C++.

For the second part of the project, …

# Approach

The overall approach to this project was two-fold. For the first part, the provided all\_pairs Codeblocks project was used as a reference to create OpenMP versions of all the functions within the project. These functions were then measured for performance on various sizes of the overall matrix that they would compute. The methodologies provided in the project were compared against their OpenMP counterparts.

For the second part of the project, …

**Hardware Used:**

The project was all performed on my home computer with the following specifications:

* **CPU**

Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz

AVX2 (256-bit MM registers)

10 cores / 20 threads

20 MB Intel Smart Cache (L3-cache)

* **RAM**

32 GB DDR4 RAM

* **GPU**

TUF RTX3080 (Ampere GPU)

8704 CUDA cores  
 5 MB of L2-Cache

10GB GDDR6X

* **OS**

Windows

# Implementation

**Section 1: OpenMP All\_Pair\_Distance Implementation**

* 1. ***Code Overview***

The following protoypes were rewritten using OpenMP:

1. block\_all\_pairs (block work distribution)
2. block\_ cyclic\_all\_pairs (block cyclic work distribution)
3. dynamic\_all\_pairs (dynamic work distribution)

The three functions were re-written to utilize OpenMP to compute the pair-wise distance matrix of MNIST train images. All algorithms were tested using 12 threads and a chunk size of 2 (if the algorithm used a chunk size). A couple of modifications were made to the signatures of these functions for ease of use.

First, the following function type was declared:

typedef void (\*AllPairsWorker\_t)(

    std::vector<float> &,

    std::vector<float> &,

    uint64\_t,

    uint64\_t,

    uint64\_t);

This allows for simplistic printing and testing of functions, as the new OpenMP functions share the same type as the C++ ones. The the following helper:

void printAndTimePairs(

    std::string name,

    AllPairsWorker\_t Worker,

    AllPairWorkerData\_t \*data)

{

    std::cout << name << "...\n";

    StartTimer();

    Worker(data->mnist, data->allPairs, data->rows, data->threads, data->chunksize);

    std::cout << "\t " << name << " time = " << StopTimer() << "\n";

    std::cout << "\tall\_pair[1000] = " << data->allPairs[1000] << "\n\n";

}

The data is in the following struct, which gathers all relevant information:

typedef struct

{

    std::vector<float> mnist;

    std::vector<float> allPairs;

    uint64\_t rows;

    uint64\_t threads;

    uint64\_t chunksize;

} AllPairWorkerData\_t;

The following code defined the all\_pairs function, so that it could easily be called in a loop to test all the various sizes in a single run.

int all\_pairs(uint64\_t nRows = 60000)

{

    std::cout << "Load MNIST train-image dataset ......\n";

    StartTimer();

    std::vector<float> mnist(ROWS \* COLS, 5); // values initialized to 5

    load\_binary(mnist.data(), ROWS \* COLS,

                "./data/train-images.bin");

    StopTimer();

    // validate data

    if ((int)(mnist[156] \* 10000) != 4941) // the value should be 0.494118

        return 0;

    if (nRows < 1 || nRows > ROWS)

        return -1;

    std::vector<float> all\_pair(nRows \* nRows);

    AllPairWorkerData\_t data;

    data.allPairs = all\_pair;

    data.mnist = mnist;

    data.rows = nRows;

    data.threads = 12;

    data.chunksize = 2;

    std::cout << "\n\nCompute pair\_wise\_distance for first " << nRows << " MNIST train images (gcc) using " << data.threads << " threads \n\n";

    printAndTimePairs("block\_all\_pairs", block\_all\_pairs, &data);

    printAndTimePairs("block\_cyclic\_all\_pairs", block\_cyclic\_all\_pairs, &data);

    printAndTimePairs("dynamic\_all\_pairs", dynamic\_all\_pairs, &data);

    printAndTimePairs("OpenMP\_block\_all\_pairs", OpenMP\_block\_all\_pairs, &data);

    printAndTimePairs("OpenMP\_block\_cyclic\_all\_pairs", OpenMP\_block\_cyclic\_all\_pairs, &data);

    printAndTimePairs("OpenMP\_dynamic\_all\_pairs", OpenMP\_dynamic\_all\_pairs, &data);

    return 0;

}

Lastly, all three implementations shared the same following code to run the function. The difference between the function was in the OpenMP configuration that was performed.

void OpenMP\_CalculatePairWiseDistance(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t unused = 0)

{

#pragma omp parallel for schedule(runtime)

    for (uint64\_t i = 0; i < rows; i++)

    {

        for (uint64\_t I = 0; I <= i; I++)

        {

            float accum = float(0);

            for (uint64\_t j = 0; j < COLS; j++)

            {

                float residue = mnist[i \* COLS + j] - mnist[I \* COLS + j];

                accum += residue \* residue;

            }

            all\_pair[i \* rows + I] = all\_pair[I \* rows + i] = accum;

        }

    }

}

In this block of code, the OpenMP preprocessor command *schedule* performs an optimization based on which OpenMP schedule type was selected. The configuration for this is called with omp\_set\_schedule[1] in each respective implementation. The loop code is the same as the sequential all\_pair implementation; the OpenMP library handles all the parallelization and threads.

* 1. **OpenMP*\_block\_all\_pairs***

**Timing Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Matrix Size | 400 | 800 | 10,000 | 20,000 | 30,000 | 60,000 |
| C++ Block | 0.0062947 | 0.0207421 | 4.75428 | 24.9631 | 76.2645 | 351.505 |
| OpenMP block | 0.0023593 | 0.0034301 | 2.59424 | 18.5267 | 56.6752 | 326.576 |

**Implementation**

// block work distribution

void OpenMP\_block\_all\_pairs(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t num\_threads = 64,

    uint64\_t unused = 0)

{

    uint64\_t chunkSize = rows / num\_threads;

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(num\_threads);

    omp\_set\_schedule(omp\_sched\_static, chunkSize);

    OpenMP\_CalculatePairWiseDistance(mnist, all\_pair, rows);

}

}

Here we set the dynamic threading of OpenMP to 0 as we want to run each test with 12 threads (This is the same across all three implementations). Second, the number of threads is set. Third, the schedule is chosen as static and chunk size set to the rows / threads which is analogous to the original block\_all\_pairs implementation. Lastly, the helper function is called which computes the distance using a block work distribution.

* 1. **OpenMP*\_block\_cyclic\_all\_pairs***

**Timing Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Matrix Size | 400 | 800 | 10,000 | 20,000 | 30,000 | 60,000 |
| C++ block-cyclic | 0.0040413 | 0.0178878 | 1.86041 | 8.58314 | 24.1807 | 153.532 |
| OpenMP block-  cyclic | 0.0009989 | 0.0026154 | 1.19309 | 5.77042 | 13.2321 | 162.987 |

**Implementation**

// block cyclic work distribution

void OpenMP\_block\_cyclic\_all\_pairs(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t num\_threads = 64,

    uint64\_t unused = 0)

{

    uint64\_t chunkSize = 2;

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(num\_threads);

    omp\_set\_schedule(omp\_sched\_static, chunkSize);

    OpenMP\_CalculatePairWiseDistance(mnist, all\_pair, rows);

}

First, the number of threads is set. Second, the schedule is chosen as static and chunk size set to the 2 which is analogous to the original block\_cyclic\_all\_pairs implementation. Lastly, the helper function is called which computes the distance using a block cyclic work distribution.

* 1. **OpenMP*\_dynamic\_all\_pairs***

**Timing Results**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Matrix Size | 400 | 800 | 10,000 | 20,000 | 30,000 | 60,000 |
| C++ dynamic | 0.0038365 | 0.0123817 | 2.77072 | 9.13156 | 21.7323 | 124.304 |
| OpenMP dynamic | 0.0007513 | 0.0024572 | 0.67911 | 5.24815 | 11.8631 | 143.983 |

**Implementation**

void OpenMP\_dynamic\_all\_pairs(

    std::vector<float> &mnist,

    std::vector<float> &all\_pair,

    uint64\_t rows,

    uint64\_t num\_threads = 64,

    uint64\_t chunk\_size = 64 / sizeof(float))

{

    uint64\_t chunkSize = 2;

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(num\_threads);

    omp\_set\_schedule(omp\_sched\_dynamic, chunkSize);

    OpenMP\_CalculatePairWiseDistance(mnist, all\_pair, rows);

}

First, the number of threads is set. Second, the schedule is chosen as dynamic and chunk size set to the 2 which is analogous to the original dynamic\_all\_pairs implementation. Lastly, the helper function is called which computes the distance using a dynamic work distribution.

* 1. **Conclusion**

The OpenMP library provides optimization that outperforms manual C++ implementation of the same work distribution schemes. Though the improvement is marginal for large N, smaller work cases see significant performance hikes.

**Section 2: Celebrity Face Analysis using PyTorch**

# 5 Contributions

Blah-

# **References**

[1] OPENMP API Specification: Version 5.0 November 2018

https://www.openmp.org/