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**

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 with 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 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 bulk of the all\_pairs function, so that it could easily be called in a loop to test all the various sizes in a single run.

    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);

1.1 **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 | 0.0063072 | 0.0198152 | 3.99121 | 20.6523 | 64.2672 | 308.235 |

**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)

{

    auto block = [&](const uint64\_t &id) -> void

    {

        // pre-compute offset and stride

        uint64\_t chunk\_size = (rows + num\_threads - 1) / num\_threads;

        const uint64\_t off = id \* chunk\_size;

        const uint64\_t str = num\_threads \* chunk\_size;

// for each block of size chunk\_size in cyclic order

        for (uint64\_t lower = off; lower < rows; lower += str)

        {

            // compute the upper border of the block (exclusive)

            const uint64\_t upper = std::min(lower + chunk\_size, rows);

// for all entries below the diagonal (i'=I)

            #pragma omp parallel for

            for (uint64\_t i = lower; i < upper; i++)

            {

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

                {

                    // compute squared Euclidean distance

                    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;

                    }

                    // write Delta[i,i'] = Delta[i',i]

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

                }

            }

        }

    };

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

    #pragma omp parallel

    {

        block(omp\_get\_thread\_num());

    }

}

1.2 **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 | .0009384 | 0.0044919 | 0.803083 | 6.92754 | 19.6699 | 82.253 |

**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 chunk\_size = 64 / sizeof(float))

{

    auto block\_cyclic = [&](const uint64\_t &id) -> void

    {

        // pre-compute offset and stride

        const uint64\_t off = id \* chunk\_size;

        const uint64\_t str = num\_threads \* chunk\_size;

        // for each block of size chunk\_size in cyclic order

        for (uint64\_t lower = off; lower < rows; lower += str)

        {

            // compute the upper border of the block (exclusive)

            const uint64\_t upper = std::min(lower + chunk\_size, rows);

            // for all entries below the diagonal (i'=I)

            #pragma omp parallel for

            for (uint64\_t i = lower; i < upper; i++)

            {

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

                {

                    // compute squared Euclidean distance

                    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;

                    }

                    // write Delta[i,i'] = Delta[i',i]

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

                }

            }

        }

    };

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

    #pragma omp parallel

    {

        block\_cyclic(omp\_get\_thread\_num());

    }

}

1.3 **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 | 0.0008094 | 0.0024631 | 0.680631 | 4.68086 | 15.5662 | 106.767 |

**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))

{

    // declare mutex and current lower index

    uint64\_t global\_lower = 0;

    auto dynamic\_block\_cyclic = [&](const uint64\_t &id) -> void

    {

        // assume we have not done anything

        uint64\_t lower = 0;

        // while there are still rows to compute

        while (lower < rows)

        {

            // update lower row with global lower row

            {

                std::lock\_guard<std::mutex> lock\_guard(mutex);

                lower = global\_lower;

                global\_lower += chunk\_size;

            }

            // compute the upper border of the block (exclusive)

            const uint64\_t upper = std::min(lower + chunk\_size, rows);

            // for all entries below the diagonal (i'=I)

            for (uint64\_t i = lower; i < upper; i++)

            {

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

                {

                    // compute squared Euclidean distance

                    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;

                    }

                    // write Delta[i,i'] = Delta[i',i]

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

                }

            }

        }

    };

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

    #pragma omp parallel

    {

        dynamic\_block\_cyclic(omp\_get\_thread\_num());

    }

}

# 5 Contributions

Blah-

# 6 References

Blah