CSE 625 Parallel Programming

Project 2

By Caleb Klenda

# Machine Specifications

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

# Problem 1

1. Row Major Memory stores each row of the matrix in sequential order with the last element of the first row being adjacent in memory to the first element in the seconds row and so on. Thus, the result is a 1-D array storing the information of the 2D matrix.
2. Tmm speeds up the performance be performing a transpose on the matrix before attempting to multiply them. This is due to the fact that the locality is better than non-transposed matrices and so there are fewer cache misses on lookup. Avx\_tmm improves upon this idea further by using \_\_mm256 registers to perform SIMD operations.
3. **Timing Table Results:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Problem 1.2** | 200 | 400 | 800 | 1,600 | 3,200 | 6,400 | 12,800 |
| matrixMul\_RowMajor | 0.0158936 | 0.136338 | 1.16626 | 17.4047 | 165.364 |  |  |
|  |
| matrixMul\_tmm | 0.0153014 | 0.120813 | 0.964209 | 7.7588 | 63.4531 | 545.07 |  |  |
|  |
| Speed-up | 3.87% | 12.85% | 21% | 124% | 161% |  |  |  |
| matrixMul\_AVX\_tmm | 0.004846 | 0.0344672 | 0.261497 | 2.20101 | 19.0317 | 150.835 | 1196.39 |  |
|  |
| Speed-up | 227.97% | 396% | 446% | 791% | 869% |  |  |  |

1. Some Screenshots:

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Problem 2

2.1

For loop timing table results:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 200X200 | 400X400 | 800X800 |
| For-loop Timing | 5.39s | 41.6s | 5m 31s |

2.2

Numpy timing table results:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 200X200 | 400X400 | 800X800 | 1600X1600 | 3200X3200 | 6400X6400 | 12,800X12,800 |
| matmul Timing | 3.97ms | .991ms | 3.47ms | 16.4ms | 136ms | 1.24secs | 10.8secs |

Method to create the matrices

import numpy as np

def createMatrices(s):

    mat1 = np.random.random((s, s)).astype(np.float32)

    mat2 = np.random.random((s, s)).astype(np.float32)

    return (mat1, mat2)

For-loop multiplication implementation

import numpy as np

sizes = [200, 400, 800]

def multiplyMatrices(m1, m2, size):

    result = np.empty((size, size), dtype=float)

    for i in range(len(m1)):

        # iterate through columns of M2

        for j in range(len(m2[0])):

            # iterate through rows of M2

            for k in range(len(m2)):

                result[i][j] += m1[i][k] \* m2[k][j]

def runTimingTests():

    for size in sizes:

        print("Multplying matrices of size", size)

        mat1, mat2 = createMatrices(size)

        %time multiplyMatrices(mat1, mat2, size)

        print("Done")

runTimingTests()

NumPy multiplication implementation

sizes = [200, 400, 800, 1600, 3200, 6400, 12800]

def runNumpyTimingTests():

    for size in sizes:

        print("Multplying matrices of size", size)

        mat1, mat2 = createMatrices(size)

        %time np.matmul(mat1, mat2)

        print("Done")

runNumpyTimingTests()

Screenshots:

Text

Description automatically generated

Text

Description automatically generated

# Problem 3

Each function was tested on float vectors of the indicated size with all elements equal to 1. This made the calculation easy to determine and check if correct.

Sequential Dot-Function:

float SequentialDot(const std::vector<float> &v1, const std::vector<float> &v2)

{

    float result = 0;

    size\_t length = (v1.size() <= v2.size() ? v1.size() : v2.size());

    for (int i = 0; i < length; ++i)

    {

        result += v1[i] \* v2[i];

    }

    return result;

}

AVX Dot-function:

float AVXDot(const std::vector<float> &v1, const std::vector<float> &v2)

{

    \_\_m256 C = \_mm256\_setzero\_ps();

    size\_t length = (v1.size() <= v2.size() ? v1.size() : v2.size());

    float result;

    for (int i = 0; i < length; i += 8)

    {

        \_\_m256 X = \_mm256\_setzero\_ps();

        const \_\_m256 mmA = \_mm256\_loadu\_ps((float \*)&v1[i]);

        const \_\_m256 mmB = \_mm256\_loadu\_ps((float \*)&v2[i]);

        X = \_mm256\_mul\_ps(mmA, mmB);

        result += hsum256\_ps\_avx(X);

    }

    return result;

}

Test results in the following table:

|  |  |  |
| --- | --- | --- |
|  | 6,400,000 | 64,000,000 |
| Sequential time | 0.0206596 seconds | 0.063169 |
| AVX time | 0.0144532 seconds | 0.146274 |
| Sequential result | 6.4e+06 | 1.67772e+07 |
| AVX result | 6.4e+06 | 6.4e+07 |

The sequential dot result is not accurate for two vectors of size larger than 16,777,216. This is because 32-bit floats (according to IEEE-754) are stored in the following format: sign (1 bit) + exponent (8 bits) + mantissa (23 bits). The mantissa is where the value is stored and 16,777,216 is exactly 2^24 so any number more precice (like 16,777,217) cannot be stored in a 32-bit float. This obviously causes a calculation issue and is the reason any number higher results in the same answer of 16,777,216 because it cannot increment. The simplest solution to this problem is to use a double instead to increase the precision, though this may slow performance.

AVX is accurate as it uses 256-bit registers to store 8 elements of the overall vector at a time, each in a 32-bit slot. Since each slot is occupied by only a single number, there is not issues with precision that causes it to fail.

Some Screenshots:

Text

Description automatically generated

Text

Description automatically generated

# Problem 4

(25 points)

C++ Multi-threaded Matrix multiplication implementation

void matrixMul\_RowMajor\_threaded(float \*C, float \*A, float \*B, int RA, int CA, int CB, int num\_threads)

{

    // use lambda function

    auto multMatBlock = [&](const int& id, float \*C, float \*A, float \*B, int RA, int CA, int CB)

    {

        // compute chunk size, lower and upper for task id

        const int chunk = (RA + num\_threads-1) / num\_threads;

        const int lower = id \* chunk;

        const int upper = std::min(lower+chunk, RA);

        int row, col;

        for (row = lower; row < upper; row++)

        {

            for (col = 0; col < CB; ++col)

            {

                float Cvalue = 0;

                for (int k = 0; k < CA; k++)

                    Cvalue += A[row \* CA + k] \* B[k \* CB + col];

                C[row \* CB + col] = Cvalue;

            }

        }

    };

    std::vector<std::thread> threads;

    for (int id = 0; id < num\_threads; id++)

    {

        std::cout<< "Creating thread Id: " << id << std::endl;

        threads.emplace\_back(multMatBlock, id, C, A, B, RA, CA, CB);

    }

    for (auto& thread : threads)

    {

        thread.join();

    }

    std::cout<< "Threaded Matrix Multiplicaiton Complete" << std::endl;

}

Timing Table Results:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Timing Table | 4 threads | 8 threads | 16 threads | 20 threads | 32 threads |
| 3200x3200 | 39.9271 | 22.2554 | 16.723 | 14.8734 | 16.0783 |
| 6400x6400 | 405,894 | 261.577 | 392.952 | 196.498 | 210.766 |

Based on the timing results and system architecture, 20 threads seems to be optimal. For some reason, 16 threads was slower with the larger data set, but in both cases 20 was the highest performing. Since my machine has 20 threads to work with, 20 threads makes since to be the most optimal.