High Performance Computer Architectures Practical Course - Exercise 3 -

Tutorium 1

David Jordan (6260776)
Florian Rüffer (7454628)
Michael Samjatin (7485765)

May 9, 2023

1 Neural Networks and SIMD

1

The task is to adapt the feedForward function in the MLP.cpp file. In order for this to work the affineTransform, SoftMax and MatMul2D1D had to be adapted to the self-defined fvec datatype.

```
std::vector<fvec> MLPMath::MatMul2D1D(std::vector<std::vector<float>>& A, std::vector<fvec>& B) {

    std::size_t nArows = A.size();
    std::size_t nAcolumns = A[0].size();
    std::size_t nBrows = B.size();

    // initialize matrix C to all zero values
    std::vector<fvec> C(nArows);

    // check if A and B are commensurate and multiply
    if (nAcolumns == nBrows) {
        // --------------------------
        // loop over all rows of the matrix A
        for(std::size_t j = 0; j < nArows; j++){
            // declare an fvec to store the intermediate results in
            fvec tempo = fvec(0.0f);
            // store the j-th row of A in another temporary variable
            std::vector<float>& tmp = A[j];
            // looping over the entries and multiplying
            for (std::size_t l = 0; l < nAcolumns; l++){
                // utlizing the fvec multiplication for SIMD
                tempo += tmp[l] * B[l];
            }
            C[j] = tempo;
        }

        // --------------------------
        return C;
    }
    else {
        std::cerr << "Matrix indexes incompatible for multiplication. A columns =" << nAcolumns
```

```
                << "B rows =" << nBrows << std::endl;
            exit(EXIT_FAILURE);
        }
}

// The function applies the softmax activation function to an input vector of type fvec
std::vector<fvec> MLPMath::applySoftmax(std::vector<fvec>& input) {

    int inputSize = (int)input.size();
    // The output vector is initialised to be of the same size as the input vector
    std::vector<fvec> output(inputSize);
    // A vector to store the exponential values for each input vector
    std::vector<float> exponentialValue(inputSize);
    // Stores the sum of the exp values
    float exponentialSum = 0.0f;

    // ------------------------------
    // Each column of the imput matrix is looped over
    for(int i = 0; i < fvecLen; i++){
        // determines the current max value in the selected column
        float maximumV = input[0][i];
        for (std::size_t j = 1; j< inputSize; j++){
            if (maximumV < input[j][i]){
                maximumV = input[j][i];
            }
        }

        // Determine the exponential values, first clearing the values
        exponentialSum = 0.0f;
        exponentialValue.clear();
        exponentialValue.resize(inputSize);

        // For each input vector in the selected column calculate the exponential value
        for (std::size_t k = 0; k<inputSize; k++){
            float temp = exp(input[k][i] - maximumV);
            exponentialValue[k] = temp;
            exponentialSum += temp;
        }

```

```
        // Each exp value is divided by the sum of all exponential values to calculate the softmax output
        for (std::size_t l = 0; l<inputSize; l++){
            output[l][i] = exponentialValue[l] / exponentialSum;
        }
    }
    // --------------------------

    return output;
}

std::vector<fvec> MLPMath::affineTransform(std::vector<std::vector<float>>& weight, std::vector<fvec>& input, std::vector<float>& bias) {

    std::vector<fvec> result(bias.size());
    std::vector<fvec> term1 = MLPMath::MatMul2D1D(weight, input);

    for (std::size_t i = 0; i < bias.size(); i++) {
        result[i] = term1[i] + bias[i];
    }

    return result;
}
```

File 1: SIMDed math functions

The functions now utilize the SIMDed arithmetic operators of the fvec class.

2

The task for this exercise is to add SIMD (Single Instruction, Multiple Data) support to our neural network functions. We will start of with the simd-ized version of the ReLU activation function.

The first noticable detail is the changed input. We input type is "fvec". Fvec is a SIMD-packed set of values, to be exact 4 packed 32-bit floating point numbers. Once again we create a result vector, the same size as the input. Subsequently we loop for 1.) every element of fvec (here this is 4) and 2.) the number of elements of the input. As a result, we loop over every element of the input.

Finally, we check the conditions associated with the ReLU activation function for every of those elements. The defintion of ReLU is provided in the Appendix

(5.1)

```
    std::vector<fvec> MLPMath::applyReLU(std::vector<fvec>& input) {

    std::vector<fvec> result(input.size());

    for (std::size_t iv = 0; iv < fvecLen; iv++) {
        for (std::size_t i = 0; i < input.size(); i++) {
            if (input[i][iv] > 0.0f) {
                result[i][iv] = input[i][iv];
            }
            else {
                result[i][iv] = 0.0f;
            }
        }
    }

    return result;
}
```

File 2: SIMD-ized ReLU

Another essential part is the backPropActivation function. Here we provide just the code snippet, which specifically implements ReLU.

In this case there are not many significant changes. We want to ensure, that this function will execute properly if provided with a fvec vector input. For this to work, line 4 is the most essential one to understand. Here the fvec-type overloads the ">"-operator. To understand the underlying functionality, please view the following example.

"rawOutput" contains the elements: [1.0, -1.0, 2.0, -0.5]
"zero" contains the elements [0.0, 0.0, 0.0, 0.0]

Overloading will produce a result like this: $[1.0 > 0.0, -1.0 > 0.0, 2.0 > 0.0, -0.5 > 0]$

This checks the condition for every element and can the abbreviated into the following form: [0xFFFFFFFF, 0x00000000, 0xFFFFFFFF, 0x00000000] (1 - True & 0 - False)

```
1    case 1:{
2        fvec zero = fvec(0.0f);
3        for (std::size_t i = 0; i < rDelta.size(); i++) {
4            rDelta[i] = (rawOutput[i] > zero) * rDelta[i];
5        }
6        break;
7    }
```

File 3: backPropActivation

Finally we need to incorporate our SIMD-ized ReLU function into the applyActivation function (Line 10-14).

```
1      std::vector<fvec> MLPNet::applyActivation(std::vector<fvec>& input) {
2  
3      std::vector<fvec> output(input.size());
4      switch (activationType_) {
5          case 0: {// TanH
6              output = MLPMath::applyTanH(input);
7              return output;
8              break;
9          }
10         case 1: {// SIMD-ized ReLU
11             output = MLPMath::applyReLU(input);
12             return output;
13             break;
14         }
15         default: {
16             return output;
17             break;
18         }
19     }
20 }
```

File 4: applyActivation

3

When the program is run on the "mnisttrain.csv" and "mnisttest.csv" data, and the required dimensions of 10000 training / test examples, a batch size of 40 with 10 epochs and teh RELU activation function the following output is recorded:

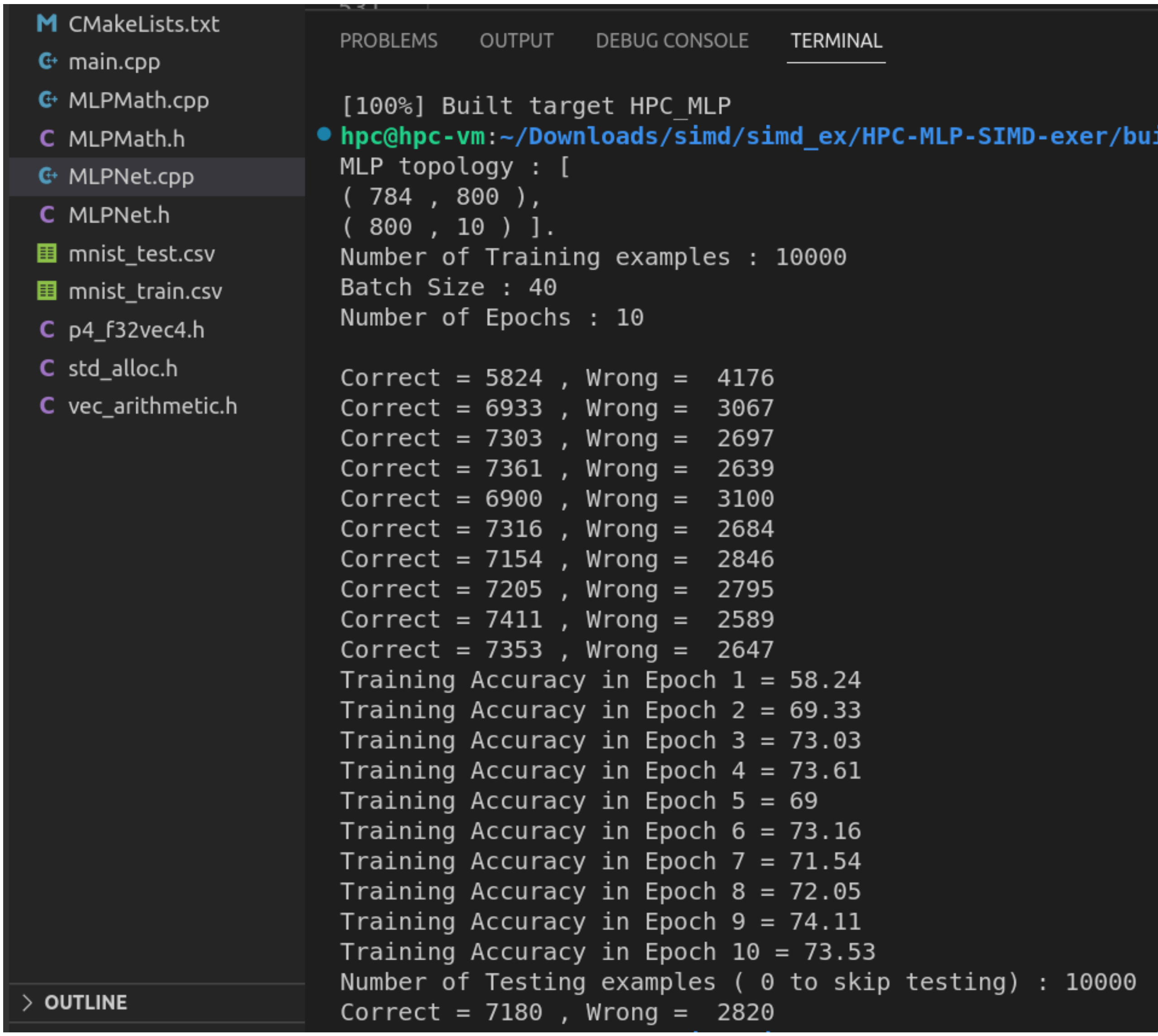


Figure 1: Output

The peak training accuracy of 74.11 in epoch 9 and a testing accuracy of 71.80 is recorded when we run the program on the data.

2 Matrix

In this task we need to speed up Matrix calculations using SIMD. As displayed below, we want to go from matrix a to matrix c.

$$a = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix}$$

$$c = \begin{bmatrix} \sqrt{a_0} & \sqrt{a_1} & \sqrt{a_2} \\ \sqrt{a_3} & \sqrt{a_4} & \sqrt{a_5} \\ \sqrt{a_6} & \sqrt{a_7} & \sqrt{a_8} \end{bmatrix}$$

First of all we need to change the input & output matrices (output for scalar computation stays the same). We do this by changing the memory alignment of the values of the matrices. This is a requirement for SIMD-ized operations.

```
float a[N][N] __attribute__((aligned(16)));       // input array
float c[N][N]; // output array for scalar computations
float c_simd[N][N] __attribute__((aligned(16))); // output array for SIMD computations
```

File 5: Matrix.cpp

The loops for the computation part remain mostly unchanged to the scalar version. The inner loop runs N-times but our loop variable increases by 4, because each vector has 4 elements and those can the calculated in parallel. We loop for $NIter - 1$ times, to neglect memory reading time. Subsequently we loop over matrix a to transform it into matrix c (NxN for rows and columns). The distinct section ranges from line 5 to line 7. We define two new vectors, called aVec and cVec, which suppose to be the vector counter parts to the scalar versions of matrix a and matrix c. At the beginning those were initialized with float values, so we use the reinterpret_cast command to allow treating a variables memory representation as another type. In this case we substitute float for fvec. Finally we just plug those values into the template function to calculate the root (Line 7).

```
1  TStopwatch timerSIMD;
2  for( int ii = 0; ii < NIter; ii++ )
3      for( int i = 0; i < N; i++ ) {
4          for( int j = 0; j < N; j++ ) {
5              fvec &aVec = reinterpret_cast<fvec&>(a[i][j]);
6              fvec &cVec = reinterpret_cast<fvec&>(c_simd[i][j]);
7              cVec = f(aVec);
8          }
9      }
10 timerSIMD.Stop();
```

File 6: Matrix.cpp

We can compute 4 values in parallel and therefore can expect a speed-up factor of 4. Due to running environment deviation, the speed-up factor varies.

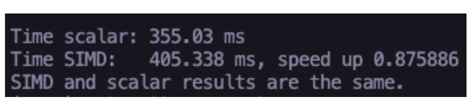


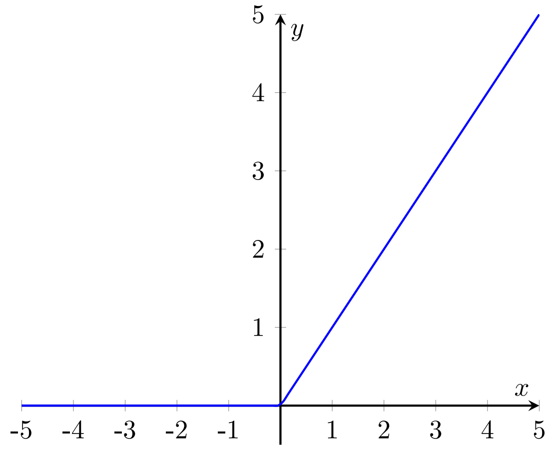
Figure 2: Output

3 Quadratic Equation

4 CheckSum

5 Appendix

5.1 ReLU



$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Figure 3: Add caption