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.

```
1  std::vector<fvec> MLPMath::MatMul2D1D(std::vector<std
       ::vector<float>>& A, std::vector<fvec>& B) {
2
3      std::size_t nArows = A.size();
4      std::size_t nAcolumns = A[0].size();
5      std::size_t nBrows = B.size();
6
7      // initialize matrix C to all zero values
8      std::vector<fvec> C(nArows);
9
10     // check if A and B are commensurate and multiply
11     if (nAcolumns == nBrows) {
12         //
               ----------------------------------------------------------
13         // loop over all rows of the matrix A
14         for(std::size_t j = 0; j < nArows; j++){
15             // declare an fvec to store the
                   intermediate results in
16             fvec tempo = fvec(0.0f);
17             // store the j-th row of A in another
                   temporary variable
18             std::vector<float>& tmp = A[j];
19             // looping over the entries and
                   multiplying
20             for (std::size_t l = 0; l < nAcolumns; l
                   ++){
21                 // utlizing the fvec multiplication
                       for SIMD
22                 tempo += tmp[l] * B[l];
23             }
24             C[j] = tempo;
25         }
26
27         //
               ----------------------------------------------------------
28         return C;
```

```
29      }
30      else {
31          std::cerr << "Matrix indexes incompatible for
               multiplication. A columns =" << nAcolumns
               << "B rows =" << nBrows << std::endl;
32          exit(EXIT_FAILURE);
33      }
34  }
35
36  // The function applies the softmax activation
       function to an input vector of type fvec
37  std::vector<fvec> MLPMath::applySoftmax(std::vector<
       fvec>& input) {
38
39      int inputSize = (int)input.size();
40      // The output vector is initialised to be of the
           same size as the input vector
41      std::vector<fvec> output(inputSize);
42      // A vector to store the exponential values for
           each input vector
43      std::vector<float> exponentialValue(inputSize);
44      // Stores the sum of the exp values
45      float exponentialSum = 0.0f;
46
47      //
           ------------------------------------------------------------
48      // Each column of the imput matrix is looped over
49      for(int i = 0; i < fvecLen; i++){
50          // determines the current max value in the
               selected column
51          float maximumV = input[0][i];
52          for (std::size_t j = 1; j< inputSize; j++){
53              if (maximumV < input[j][i]){
54                  maximumV = input[j][i];
55              }
56          }
57
58          // Determine the exponential values, first
               clearing the values
59          exponentialSum = 0.0f;
60          exponentialValue.clear();
61          exponentialValue.resize(inputSize);
62
63          // For each input vector in the selected
               column calculate the exponential value
```

```
64          for (std::size_t k = 0; k<inputSize; k++){
65              float temp = exp(input[k][i] - maximumV);
66              exponentialValue[k] = temp;
67              exponentialSum += temp;
68          }
69
70          // Each exp value is divided by the sum of all
                exponential values to calculate the
               softmax output
71          for (std::size_t l = 0; l<inputSize; l++){
72              output[l][i] = exponentialValue[l] /
                   exponentialSum;
73          }
74      }
75      //
            ----------------------------------------------------------


76
77      return output;
78  }
79
80  std::vector<fvec> MLPMath::affineTransform(std::vector
        <std::vector<float>>& weight, std::vector<fvec>&
        input, std::vector<float>& bias) {
81
82      std::vector<fvec> result(bias.size());
83      std::vector<fvec> term1 = MLPMath::MatMul2D1D(
            weight, input);
84
85      for (std::size_t i = 0; i < bias.size(); i++) {
86          result[i] = term1[i] + bias[i];
87      }
88
89      return result;
90  }
```

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)

```
1  std::vector<fvec> MLPMath::applyReLU(std::vector<
       fvec>& input) {
2
3      std::vector<fvec> result(input.size());
4
5      for (std::size_t iv = 0; iv < fvecLen; iv++) {
6          for (std::size_t i = 0; i < input.size(); i++)
               {
7              if (input[i][iv] > 0.0f) {
8                  result[i][iv] = input[i][iv];
9              }
10             else {
11                 result[i][iv] = 0.0f;
12             }
13         }
14     }
15
16     return result;
17 }
```

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:

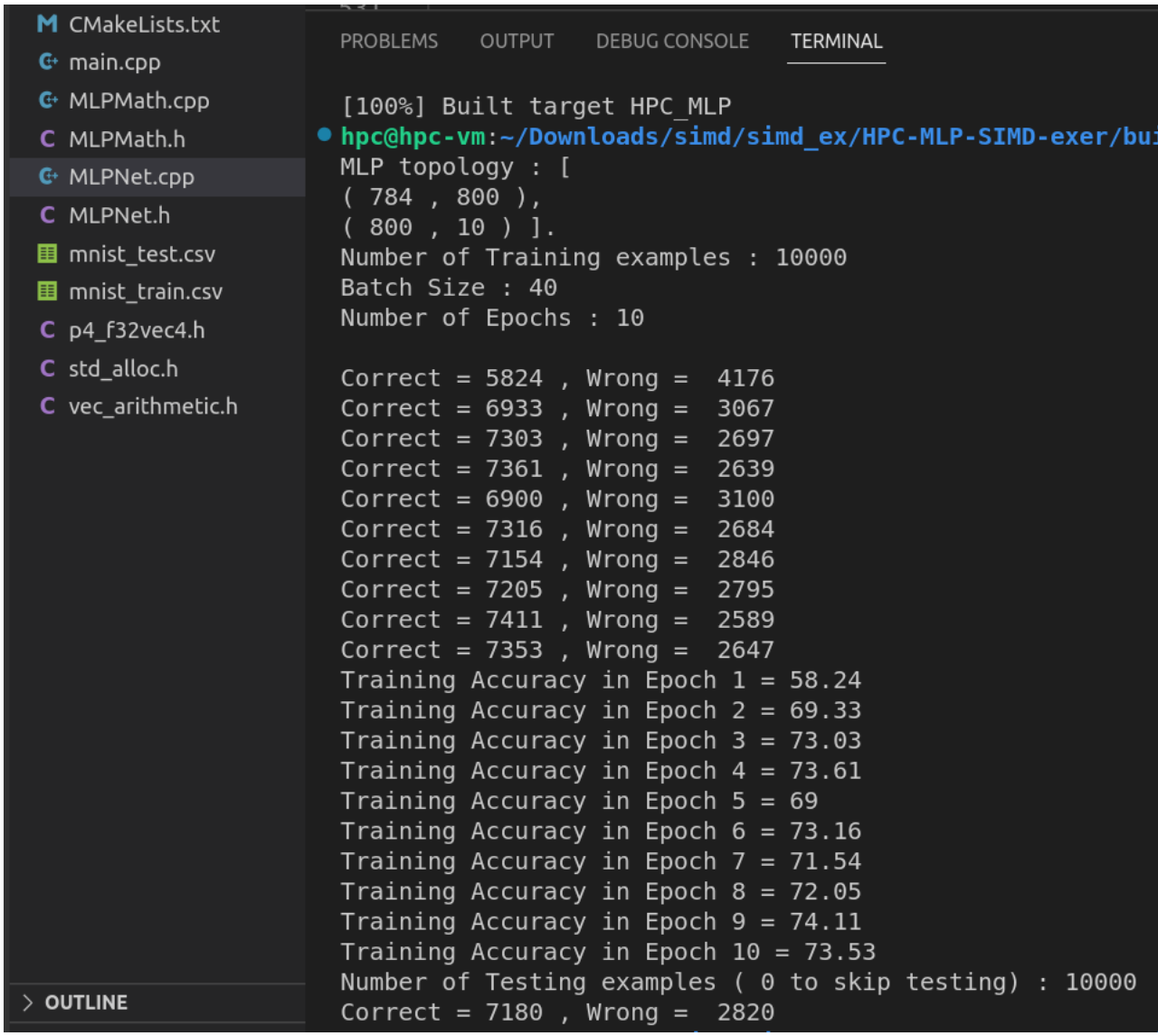


Figure 1: Output

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.

```
1 float a[N][N] __attribute__((aligned(16)));       //
    input array
2 float c[N][N]; // output array for scalar computations
3 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.

```
Time scalar: 355.03 ms
Time SIMD:   405.338 ms, speed up 0.875886
SIMD and scalar results are the same.
```

Figure 2: Output

3 Quadratic Equation

3.1 Quadratic Equation - Copying of data, SIMD

In this task we first need to create a copy of the data into a __m128 datatype. For this, we use the function _mm_set_ps. This function takes 4 float values and concating them in mirrored order. Now we need to calculate the root - here we use the predefined functions (_mm_div_ps, ...).

3.2 Quadratic Equation - Casting, SIMD

Here there is not much different to the exercise above. We need to cast the data into a __m128 datatype with the reinterpret_cast function. Then we can again use the predefined functions to calculate the root like before.

3.3 Quadratic Equation - Copying of data, fvec

We first copy the data with defining the fvecs (takes 4 arguments - 32 bit). Then we calculate as usual.

3.4 Quadratic Equation - Casting, fvec

In this exercise we need to cast the data like in one of the previous exercises. We did this with the reinterpret_cast function. Then we can calculate the root as usual.

3.5 Quadratic Equation - Output

Time scalar: 75.4349 ms
Time SIMD1: 34.869 ms, speed up 2.16338
Time SIMD2: 34.0381 ms, speed up 2.21619
Time SIMD3: 35.1319 ms, speed up 2.14719
Time SIMD4: 35.0568 ms, speed up 2.15179
SIMD1 and scalar results are the same.
SIMD2 and scalar results are the same.
SIMD3 and scalar results are the same.
SIMD4 and scalar results are the same.

3.6 Quadratic Equation - Time

The data computed is for all the options the same - 10000 vectors. Theoretically the times should be quartered, but it differs a bit.

4 CheckSum

To get the checksum, we need to XOR over all the bytes of the array. If we want to use the SIMD-ized version, we need to XOR over all in parallel. To do this, we first need to map the array to fvecs - reinterpret_cast. This datatype is needed for the XOR with SIMD. Then we can use the template command to XOR over all the bytes in parallel. The result is a fvec, which we need to map back to a char. This is done by using the reinterpret_cast command again. After we iterated over the resulting char-array and XORed over all the bytes, we get the checksum. Now we can stop the timer.

If we want to parallelize without SIMD, we map the the array to integers (4:1). This array is XORED and we have a vector with 4 checksums. To get the final checksum, we need to XOR over all the 4 checksums. And again, we stop the time.

4.1 CheckSum-Result

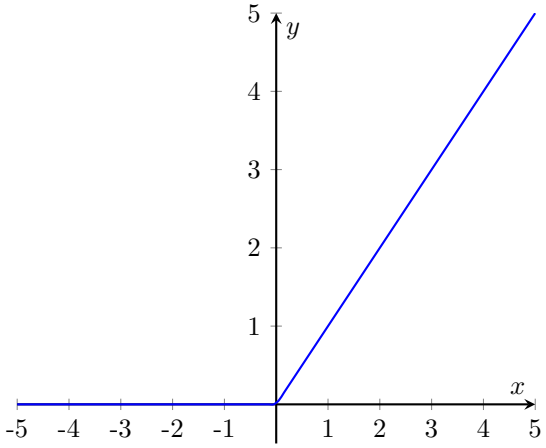
The speedup we've got with the Time SIMD is only 8.32 - in theory it should be close to 16 (because of the parallel computations), but in reality there is a bit of difference. The speed up of the Scalar Integer is about 4 (3.91) and pretty much what we expected - 4 bytes per integer.

4.2 CheckSum-Output

Time scalar: 112.757 ms
Time INT: 28.821 ms, speed up 3.91232
Time SIMD: 13.551 ms, speed up 8.32094
Results are the same.

5 Appendix

5.1 ReLU



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