High Performance Computer Architectures
Practical Course
- Exercise 3 -

May 3, 2023

Neural Networks and SIMD

This week, there are several tasks on SIMD instructions in C++. The MLP implementation of the previous week should be SIMDized over the batches. Furthermore, there are old exercises to solve.

We will now use SIMD in our neural network package. You might have noticed that all the examples in a batch can be run through the neural network independently. It is only at the end of a batch that we sum up all the gradients and update the weights. So we can pack together fvecLen number of training examples of a batch into a SIMD vector and run them through the neural network together.

Problem

Follow the comments in the code to complete the SIMD-ized neural network:

1. Complete the `feedForward()` function in MLPNet.cpp. You will need to write SIMD-ized `affineTransform`, `Softmax` and `MatMul2D1D` functions for this.

2. Implement a SIMD-ized ReLU activation as in case 1 of `applyActivation()` function in MLPNet.cpp. The ReLU activation function is defined as

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0 & \text{otherwise} \end{cases} . \tag{1}$$

When the argument to ReLU is a vector, it is applied element-wise. Also write the back-propagation for ReLU using

$$ReLU'(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise} \end{cases} . \tag{2}$$

3. Run the network for topology = $\{784, 10\}$, 10000 training examples, 10000 testing examples, batch size = 40, number of epochs = 10, activation function ReLU. Write down the peak training and testing accuracy.

Part of the source code of P4_F32vec4.h

```
44  class F32vec4
45  {
46   public:
47
48    __m128 v;
49
50    float & operator[]( int i ){ return (reinterpret_cast<float*>(&v))[i]; }
51    float   operator[]( int i ) const { return (reinterpret_cast<const
    float*>(&v))[i]; }
52
53    F32vec4( ):v(_mm_set_ps1(0)){}
54    F32vec4( const __m128 &a ):v(a) {}
55    F32vec4( const float &a ):v(_mm_set_ps1(a)) {}
56
57    F32vec4( const float &f0, const float &f1, const float &f2, const float
    &f3 ):v(_mm_set_ps(f3,f2,f1,f0)) {}
…
62    /* Arithmetic Operators */
63    friend F32vec4 operator +(const F32vec4 &a, const F32vec4 &b) { return
    _mm_add_ps(a,b); }
…
72    /* Square Root */
73    friend F32vec4 sqrt ( const F32vec4 &a ){ return _mm_sqrt_ps (a); }
…
214 } __attribute__ ((aligned(16)));
215
216
217 typedef F32vec4 fvec;
218 typedef float  fscal;
219 const int fvecLen = 4;
```

Headers with vector instructions emulated by scalar operations are provided in **PSEUDO_F32vec4.h** (vector with 4 entires) and **PSEUDO_F32vec1.h** (vector with 1 entry). They can be used in the similar way as **P4_F32vec4.h**, but won't give any speed up. They supposed to be used for debugging and comparison.

2_SIMD/0_Matrix: description

The Matrix exercise requires to parallelize square root extraction over a set of float variables arranged in a square matrix, see Fig. 1. The initial code gives an implemented scalar part and lives blank space for a vector part. Therefore the initial output shows 0 time for vector calculations and infinite speed up factor, which should be currently ignored.

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \qquad c = \begin{bmatrix} \sqrt{a_{00}} & \sqrt{a_{01}} & \sqrt{a_{02}} \\ \sqrt{a_{10}} & \sqrt{a_{11}} & \sqrt{a_{12}} \\ \sqrt{a_{20}} & \sqrt{a_{21}} & \sqrt{a_{22}} \end{bmatrix} = ?$$

Fig. 1. The explanation of Matrix task.

Part of the source code of Matrix.cpp

```
23  float a[N][N]; // input array
24  float c[N][N]; // output array for scalar computations
25  float c_simd[N][N]; // output array for SIMD computations
```

Part of the source code of Matrix.cpp

```
26
27  template<typename T> // required calculations
28  T f(T x) {
29    return sqrt(x);
30  }
...
45  int main() {
46
47      // fill classes by random numbers
48    for( int i = 0; i < N; i++ ) {
49      for( int j = 0; j < N; j++ ) {
50        a[i][j] = float(rand())/float(RAND_MAX); // put a random value, from 0 to
    1
51      }
52    }
53
54      /// -- CALCULATE --
55      /// SCALAR
56    TStopwatch timerScalar;
57    for( int ii = 0; ii < NIter; ii++ )
58      for( int i = 0; i < N; i++ ) {
59        for( int j = 0; j < N; j++ ) {
60          c[i][j] = f(a[i][j]);
61        }
62      }
63    timerScalar.Stop();
64
65      /// SIMD VECTORS
66    TStopwatch timerSIMD;
67      // TODO
68    timerSIMD.Stop();
69
70    double tScal = timerScalar.RealTime()*1000;
71    double tSIMD1 = timerSIMD.RealTime()*1000;
72
73    cout << "Time scalar: " << tScal << " ms " << endl;
74    cout << "Time SIMD:   " << tSIMD1 << " ms, speed up " << tScal/tSIMD1 <<
    endl;
75
76    CheckResults(c,c_simd);
77
78    return 1;
79  }
```

Typical output

```
Time scalar: 599.968 ms
Time SIMD:   0 ms, speed up inf
ERROR! SIMD and scalar results are not the same.
```

2_SIMD/0_Matrix: solution

For vector calculations all N*N scalar variables need to be divided into N*(N/4) groups with 4 elements, that will be treated as vectors. The solution groups scalars within each row, thus creating a matrix with N rows and N/4 columns.

Initial data copying into vectors can be avoided by reinterpreting them as vectors in case the scalar data is aligned to 16 bytes (line 23 of the solution). In the same way copying can be avoided at the output stage (line 25).

Although one can use the same algorithm structure with a triple loop as in the scalar code, we iterate over 4 elements groups rather than over each matrix element, therefore the step of the innermost loop need to be modified.

Part of the source code of Matrix_solution.cpp

```
23  float a[N][N] __attribute__((aligned(16)));      // input array
24  float c[N][N];                                   // output array for scalar
    computations
25  float c_simd[N][N] __attribute__((aligned(16))); // output array for SIMD
    computations
…
66    TStopwatch timerSIMD;
67    for( int ii = 0; ii < NIter; ii++ )
68      for( int i = 0; i < N; i++ ) {
69        for( int j = 0; j < N; j+=fvecLen ) {
70            fvec &aVec = reinterpret_cast<fvec&>(a[i][j]);
71            fvec &cVec = reinterpret_cast<fvec&>(c_simd[i][j]);
72            cVec = f(aVec);
73        }
74      }
75    timerSIMD.Stop();
```

Typical output after solution

```
Time scalar: 604.324 ms
Time SIMD:   150.4 ms, speed up 4.01811
SIMD and scalar results are the same.
```

Since the function **f(…)**, which must be applied to the data, is a template, we can use the same function call format.

Since 4 float variables fit into a single SIMD vector, all calculations are done in parallel and no overhead operations are required, the expected speed-up factor should be 4.

2_SIMD/1_QuadraticEquation: description

The QuadraticEquation exercise requires to vectorize solution of a set of quadratic equations in four different ways: using either (1) copying of data or (2) casting and using either (3) SIMD intrinsics or (4) **fvec** type from the header file. Then (5) compare the calculation time depending on amount of processed data.

It is recommended to compile the code with **-fno-tree-vectorize** option to prevent auto-vectorization of the scalar code, otherwise comparison of the vectorized and scalar codes will not be direct.

Part of the source code of QuadraticEquation.cpp

```
74    // fill parameters by random numbers
75    for( int i = 0; i < N; i++ ) {
76      a[i] = float(rand())/float(RAND_MAX); // put a random value, from 0 to 1
```

Part of the source code of QuadraticEquation.cpp

```
77      b[i] = float(rand())/float(RAND_MAX);
78      c[i] = -float(rand())/float(RAND_MAX);
79    }
80
81      /// -- CALCULATE --
82
83    // scalar calculations
84    TStopwatch timerScalar;
85    for(int io=0; io<NIterOut; io++)
86      for(int i=0; i<N; i++)
87      {
88        float det = b[i]*b[i] - 4*a[i]*c[i];
89        x[i] = (-b[i]+sqrt(det))/(2*a[i]);
90      }
91    timerScalar.Stop();
```

Typical output

```
Time scalar: 431.702 ms
Time SIMD1:   0 ms, speed up inf
Time SIMD2:   0.000953674 ms, speed up 452672
Time SIMD3:   0 ms, speed up inf
Time SIMD4:   0.000953674 ms, speed up 452672
ERROR! SIMD1 and scalar results are not the same.
ERROR! SIMD2 and scalar results are not the same.
ERROR! SIMD3 and scalar results are not the same.
ERROR! SIMD4 and scalar results are not the same.
```

2_SIMD/1_QuadraticEquation: solution

(1) The first task is to vectorize calculations using copying of data and SIMD intrinsics.

One need to use **__m128** type and **_mm_set_ps** function to copy the data. This function takes four float arguments and fills 4 32-bit elements in **__m128** variable starting from the last one. Therefore for i-th vector one uses (i*fvecLen+3)-th, (i*fvecLen+2)-th, (i*fvecLen+1)-th and (i*fvecLen)-th scalars as arguments.

The **_mm_set_ps1** function is used to convert constants 4 and 2 into the SIMD type. The **_mm_sub_ps**, **_mm_mul_ps**, **_mm_sqrt_ps** and **_mm_div_ps** are used for all required calculations.

Part of the source code of QuadraticEquation_solution.cpp

```
98        __m128 aV = _mm_set_ps(a[i*fvecLen+3],a[i*fvecLen+2],a[i*fvecLen
      +1],a[i*fvecLen]);
99        __m128 bV = _mm_set_ps(b[i*fvecLen+3],b[i*fvecLen+2],b[i*fvecLen
      +1],b[i*fvecLen]);
100       __m128 cV = _mm_set_ps(c[i*fvecLen+3],c[i*fvecLen+2],c[i*fvecLen
      +1],c[i*fvecLen]);
101
102       const __m128 det = _mm_sub_ps(_mm_mul_ps(bV,bV) ,
      _mm_mul_ps(_mm_set_ps1(4),_mm_mul_ps(aV,cV) ) );
103       __m128 xV =
      _mm_div_ps(_mm_sub_ps(_mm_sqrt_ps(det),bV),_mm_mul_ps(_mm_set_ps1(2),aV));
```

(2) The second task is vectorisation using cast and SIMD intrinsics.

Since the data is already aligned to 16 bytes one can use **reinterpret_cast** directly to the part of the scalar input and output arrays starting from the i-th element.

The calculation part remains the same.

Part of the source code of QuadraticEquation_solution.cpp

```
114     for(int i=0; i<N; i+=fvecLen)
115     {
116       __m128& aV = (reinterpret_cast<__m128&>(a[i]));
117       __m128& bV = (reinterpret_cast<__m128&>(b[i]));
118       __m128& cV = (reinterpret_cast<__m128&>(c[i]));
119
120       __m128& xV = (reinterpret_cast<__m128&>(x_simd2[i]));
```

(3) The third task is to vectorize calculations using copying of data and headers.

With headers one uses **fvec** type and **fvec** constructor to copy the data. The constructor takes 4 float arguments and fills 4 32-bit elements starting from the first one. Therefore for the i-th vector one uses (i*fvecLen)-th, (i*fvecLen+1)-th, (i*fvecLen+2)-th and (i*fvecLen+3)-th scalars as arguments.

The calculations part remains the same.

Part of the source code of QuadraticEquation_solution.cpp

```
134       fvec aV = fvec(a[i*fvecLen],a[i*fvecLen+1],a[i*fvecLen+2],a[i*fvecLen
      +3]);
135       fvec bV = fvec(b[i*fvecLen],b[i*fvecLen+1],b[i*fvecLen+2],b[i*fvecLen
      +3]);
136       fvec cV = fvec(c[i*fvecLen],c[i*fvecLen+1],c[i*fvecLen+2],c[i*fvecLen
      +3]);
137
138       const fvec det = bV*bV - 4*aV*cV;
139       fvec xV = (-bV+sqrt(det))/(2*aV);
```

(4) The fourth task is to vectorize using cast of data and headers. It can be done as in the tasks (2) and (3).

Typical output after solution

```
Time scalar: 432.194 ms
Time SIMD1:   109.914 ms, speed up 3.93211
Time SIMD2:   105.318 ms, speed up 4.1037
Time SIMD3:   111.117 ms, speed up 3.88954
Time SIMD4:   105.588 ms, speed up 4.09321
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.
```

It is expected to have exactly the same speed up factor independently of SSE intrinsics overloading. The speed up factor of 4 should be achieved with **reinterpret_cast**. It should be slightly less in the other case, because additional time is spend on data management.

(5) The initial number of vectors is 10,000, which gives the size of arrays 10000*16 bytes = 0.16 MB. All 5 arrays should fit into the cache memory (typical size 1-20 MB), therefore the speed-up factor of 4 should be achievable with **reinterpret_cast**.

When the number of vectors is increased by a factor of 1000, the size of arrays becomes 160 MB, then the arrays will be stored in the much slower RAM memory. Therefore the speed-up factor will be less than 4. The speed-up factor should remain the same (low) also for 1,000,000 vectors.

Typical output after solution

```
Time scalar: 4322.71 ms
```

Typical output after solution

```
Time SIMD1:   1197.12 ms, speed up 3.61093
Time SIMD2:   1130.53 ms, speed up 3.82361
Time SIMD3:   1215.68 ms, speed up 3.5558
Time SIMD4:   1137.61 ms, speed up 3.79981
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.
```

2_SIMD/2_CheckSum: description

The CheckSum exercise requires to vectorize the check sum calculation for the array of data. Check sum is defined as XOR-sum over all bytes of the array (lines 29-30 in the code below). In addition, consider if parallel calculations here can be done without the hardware support of parallelization.

Part of the source code of CheckSum.cpp

```
23 template< typename T >
24 T Sum(const T* data, const int N)
25 {
26     T sum = 0;
27 
28     for ( int i = 0; i < N; ++i )
29       sum = sum ^ data[i];
30     return sum;
31 }
32 
33 int main() {
34 
35     // fill string by random values
36   for( int i = 0; i < N; i++ ) {
37     str[i] = 256 * ( double(rand()) / RAND_MAX ); // put a random value, from 0
   to 255
38   }
39 
40     /// -- CALCULATE --
41 
42     /// SCALAR
43 
44   unsigned char sumS = 0;
45   TStopwatch timerScalar;
46   for( int ii = 0; ii < NIter; ii++ )
47     sumS = Sum<unsigned char>( str, N );
48   timerScalar.Stop();
```

2_SIMD/2_CheckSum: solution

The parallelization can be achieved, since XOR operator is applied bitwise and can be applied to the SIMD register data in the same way as to **char** variable, such that we can use **__m128** type to pack 16 **char** variables into a vector and treat them simultaneously.

The input **char** array should be reinterpreted as an array of **fvec**s. Then XOR sum of **fvec**s is calculated using the given template function. The resulting **fvec** variable should be reinterpreted back as an array of 16 **char** variables and the scalar sum of 16 elements is calculated directly.

Part of the source code of CheckSum_solution.cpp

```
52    unsigned char sumV = 0;
53
54    const int fvecCharLen = fvecLen*4;
55    const int NV = N/fvecCharLen;
56
57    TStopwatch timerSIMD;
58    for( int ii = 0; ii < NIter; ii++ ) {
59      fvec sumVV = 0;
60      sumVV = Sum<fvec>( reinterpret_cast<fvec*>(str), NV );
61      unsigned char *sumVS = reinterpret_cast<unsigned char*>(&sumVV);
62
63      sumV = sumVS[0];
64      for ( int iE = 1; iE < fvecCharLen; ++iE )
65        sumV ^= sumVS[iE];
66    }
67    timerSIMD.Stop();
```

Parallelization without SIMD instructions can be achieved, since XOR operator is applied bitwise and can be applied to integer data in the same way as to SIMD data and byte data. The only difference is the length of the vector, that is 4 in case of integer.

Part of the source code of CheckSum_solution.cpp

```
71    unsigned char sumI = 0;
72
73    const int intCharLen = 4;
74    const int NI = N/intCharLen;
75
76    TStopwatch timerINT;
77    for( int ii = 0; ii < NIter; ii++ ) {
78      int sumII = Sum<int>( reinterpret_cast<int*>(str), NI );
79      unsigned char *sumIS = reinterpret_cast<unsigned char*>(&sumII);
80
81      sumI = sumIS[0];
82      for ( int iE = 1; iE < intCharLen; ++iE )
83        sumI ^= sumIS[iE];
84    }
85    timerINT.Stop();
```

Typical output after solution

```
Time scalar: 274.435 ms
Time INT:   65.3081 ms, speed up 4.20216
Time SIMD:   25.3839 ms, speed up 10.8114
Results are the same.
```

A typical speed-up with integer should be 4, as expected, since 4 bytes are packed into one integer and time for the additional loop with 3 iterations is negligible. The speed-up factor using **fvec** theoretically should be 16, practically the achieved speed-up factor is about 10 to 12.

Typical output of KFLineFitter_solution2_simd.cpp

```
 Begin
Time: 0.647068 ms
 End
```