Hendrik Kühne - 35229816

Contents

In the following, we will go into detail on how the code for a neural network from Ref. [1] can be accelerated. The corresponding code can be found on GitHub [2]. After introducing the modifications on data structures, we will show how parts of the neural network were accelerated and compare the results.

We used SIMD to accelerate the forward and backward passes of convolutional and linear layers on two hardware platforms: x86 and Arm.

1 Vectorizing the `tensor` struct

In order to parallelize the neural network, we need a way to extract vectors from the `tensor` instances that form the backbone of the network. We use different vector types, depending on the hardware platform. We choose those while keeping in mind that the default numerical type in Ref. [1] is `float`, which is a 32-Bit type. For achieving a high degree of parallelism, we want vectors with many lanes.

- On Arm, there are 64-Bit (D-type) und 128-Bit (Q-type) vectors [3]. The best compromise between the necessary accuracy and high parallelism is the `float32x4_t` Q-type, which has four lanes with 32-Bit precision.
- On x86, we define a datatype `realv` with

```
typedef float real;
typedef real realv __attribute__((vector_size(64),
                                  __may_alias__,
                                  aligned(sizeof(real))));
```

  which is a vector that comprises 16 32-Bit floating point values. This definition is added to the file `tensor.h`.

Using the appropriate preprocessor directive, the necessary headers for the corresponding intrinsics are included with

```
#ifdef __ARM_64BIT_STATE
    #include <arm_neon.h>
#else
    #include <x86intrin.h>
#endif
```

where the variable `__ARM_64BIT_STATE` is only defined on Arm-hardware. The headers `arm_neon.h` respectively `x86intrin.h` contain the vector instructions for Arm [4] or x86 [5] platforms.

We would now like to extract a vector from a `tensor` instance. We do this by casting a pointer to the memory to be vectorized as the corresponding vector type pointer and then returning the de-referenced pointer. To this end, we add two functions per architecture to `tensor.h`; for Arm, these are

```
template<typename T>
static float32x4_t& V4(T& address){return *((float32x4_t*)address);}

```

```
float32x4_t& tensor<T,N0,N1,N2,N3>::V4(idx_t i0,
                                       idx_t i1=0,
                                       idx_t i2=0,
                                       idx_t i3=0){
    tensor<T,N0,N1,N2,N3>& a = *this;
    T* address = &(a(i0,i1,i2,i3));

    return ::V4(address);
}
```

where `idx_t` is an integer type.

The function `tensor<T,N0,N1,N2,N3>::V4` returns a pointer to the first value of the desired vector by reference and `V4` then returns a vector with four lanes. These functions are analogously named `tensor<T,N0,N1,N2,N3>::V16` and `V16` on x86; they return a 16-lane vector using the same mechanism. These vectors are write-safe, meaning values can be assigned to them.

The default arguments in the function call `tensor<T,N0,N1,N2,N3>::V4` / `tensor<T,N0,N1,N2,N3>::V16` serve an important purpose: Since we are casting a pointer to obtain a SIMD-vector, we are assuming the data to be saved consecutively in memory. In general, we could thus only extract vectors from the last dimension of a tensor since only these values are saved consecutively. In the special case that, from a certain dimension onward, all dimensions of the tensor have size 1, this is not true. Consider for example the instance `tensor<real,5,5,1,1> a`. Since the last two dimensions have size one, the memory layout is actually consecutive in dimension two. The default arguments `idx_t i2 = 0` and `idx_t i3 = 0` then ensure that `a.V4(2,0)` returns a vector containing the elements `a(2,0:4,0,0)`.

Using the above definitions for vector extraction from `tensor` instances thus enables us to extract vectors from the last dimension with size `Ni` > 1.

2 Vectorizing the Convolutional and Linear Layers

As mentioned earlier, we are vectorizing the forward and backward passes in `convolution.h` and `linear.h`. To this end, we add the implementation `algo_cpu_simd` which is set as current implementation with the command line argument `-a cpu_simd`. During computation, the functions

```
void Convolution2D<maxB,IC,H,W,K,OC>::forward_simd(
            tensor<real,maxB,IC,H,W>& x, int training);
void Convolution2D<maxB,IC,H,W,K,OC>::backward_simd(
            tensor<real,maxB,OC,H-K+1,W-K+1>& gy);
void Linear<M,N,K0,K1,K2>::forward_simd(
            tensor<real,M,K0,K1,K2>& x, int training);
void Linear<M,N,K0,K1,K2>::backward_simd(
            tensor<real,M,N>& gy);
```

are called; these are the vectorized versions of their `base` counterparts. The code for both Arm and SIMD is contained in these functions; the preprocessor again chooses the right one based on the variable `__ARM_64BIT_STATE`. A code excerpt from the forward pass in linear layers is discussed in Figure 2.1.

The other functions are vectorized in the same fashion: we identify the loop to be vectorized (one which iterates over the last dimension of a tensor), re-define variables as vectors as necessary and replace the operations with the appropriate intrinsics. An overview of the vector operations we used and their corresponding intrinsics can be found in Table 2.1.

Operation	Arm	x86
$v_i = a$	`vdupq_n_f32(a)`	`_mm512_set1_ps(a)`
$v_i \to v_i + a_i b_i$	`vfmaq_f32(v,a,b)`	`_mm512_fmadd_ps(a,b,v)`
$v_i \to v_i + a_i$	`vaddq_f32(v,a)`	`_mm512_add_ps(v,a)`
$\sum_i v_i$	`vaddvq_f32(v)`	`_mm512_reduce_add_ps(v)`

Table 2.1: An overview of the vector operations we used and the corresponding intrinsics on Arm or x86 [4, 5]. All of them accept and return `float32x4_t` respectively `__m512` types.

```
float32x4_t vec;
real v;
for(idx_t i = 0;i < m;i++){
    idx_t j = 0;
    for(;j+3 < N;j+=4){
         /* We will parallelize the loop over j since we only
         have access to vectors taken from the last dimension
         of a tensor. We'll use vectors with four lanes. */
        vec = vdupq_n_f32(0);
        for(idx_t k0 = 0;k0 < K0;k0++){
            for(idx_t k1 = 0;k1 < K1;k1++){
                for(idx_t k2 = 0;k2 < K2;k2++){
                    // v += x(i,k0,k1,k2) * w(k0,k1,k2,j);
                    vec = vfmaq_f32(vec,
                                    w.V4(k0,k1,k2,j),
                                    vdupq_n_f32(x(i,k0,k1,k2)));
                        // vfma(a,b,c) = a + b * c
                }
            }
        }
        vec = vaddq_f32(vec,b.V4(j));
        y.V4(i,j) = vec;
        // y(i,j) = v + b(j);
    }
    for(;j < N;j++){
         /* remainder iterations - this is just the code from
         forward_base. */
        v = 0;
        for(idx_t k0 = 0;k0 < K0;k0++){
            for(idx_t k1 = 0;k1 < K1;k1++){
                for(idx_t k2 = 0;k2 < K2;k2++){
                    v += x(i,k0,k1,k2) * w(k0,k1,k2,j);
                }
            }
        }
        y(i,j) = v + b(j);
    }
}
```

Figure 2.1: Arm code from `void Linear::forward_simd`. As discussed in Section 1, we can only extract vectors from the last dimension of a tensor with size `Ni` > 1, there are thus two loops that are candidates for vectorization: The ones over `k2` or `j`, respectively. We choose the one over `j` since the range $0 \leq$ `k2` < 3 is smaller than our vector size. The function `float32x4_t vdupq_n_f32(float32_t value)` then writes `0` to all lanes of `vec`. `float32x4_t vfmaq_f32(float32x4_t a, float32x4_t b, float32x4_t c)` performs the fused multiply-add operation `a = a + b * c` and `vaddq_f32` performs addition. We finally write the result back to the tensor using the custom function `tensor::V4`. Since the relevant tensor dimension is not necessarily divisible by four, we execute remainder iterations afterwards using the original code.

3 Results

We first need to check that the results are indeed the same. We do this by training a network uing just one training data point (one image from the mnist dataset) for 10 epochs. The results are displayed in figures 3.1 and 3.2.

```
974000: model building starts
12828000: model building ends
12875000: loading data from data
70201000: use 1 data items out of 60000
72530000: loading data from data
83668000: use 0 data items out of 10000
83813000: training starts
115118000: Train Epoch: 1 [0/1 (0%)]        Loss: 2.286959
145269000: Train Epoch: 2 [0/1 (0%)]        Loss: 0.927964
174878000: Train Epoch: 3 [0/1 (0%)]        Loss: 0.004244
204088000: Train Epoch: 4 [0/1 (0%)]        Loss: 0.001433
233316000: Train Epoch: 5 [0/1 (0%)]        Loss: 0.002503
262510000: Train Epoch: 6 [0/1 (0%)]        Loss: 0.000037
291594000: Train Epoch: 7 [0/1 (0%)]        Loss: 0.000096
320755000: Train Epoch: 8 [0/1 (0%)]        Loss: 0.000678
349916000: Train Epoch: 9 [0/1 (0%)]        Loss: 0.000182
378985000: Train Epoch: 10 [0/1 (0%)]        Loss: 0.000268
379002000: training ends
```

Figure 3.1: The command-line output for our tests using the option `-a cpu_base`.

```
752000: model building starts
25088000: model building ends
25124000: loading data from data
99219000: use 1 data items out of 60000
101700000: loading data from data
117119000: use 0 data items out of 10000
117286000: training starts
135042000: Train Epoch: 1 [0/1 (0%)]        Loss: 2.286959
151599000: Train Epoch: 2 [0/1 (0%)]        Loss: 0.927964
167871000: Train Epoch: 3 [0/1 (0%)]        Loss: 0.004244
184137000: Train Epoch: 4 [0/1 (0%)]        Loss: 0.001433
200405000: Train Epoch: 5 [0/1 (0%)]        Loss: 0.002503
216770000: Train Epoch: 6 [0/1 (0%)]        Loss: 0.000037
233244000: Train Epoch: 7 [0/1 (0%)]        Loss: 0.000096
249680000: Train Epoch: 8 [0/1 (0%)]        Loss: 0.000678
266085000: Train Epoch: 9 [0/1 (0%)]        Loss: 0.000182
282488000: Train Epoch: 10 [0/1 (0%)]        Loss: 0.000268
282507000: training ends
```

Figure 3.2: The command-line output for our tests using the option `-a cpu_simd`.

We see that the loss values are indeed the same for all epochs, so our implementation is correct.

	Arm		x86	
	Forward	Backward	Forward	Backward
Base	6 423 000 ns	17 356 000 ns	72 391 743 ns	165 699 492 ns
SIMD	2 087 000 ns	10 883 000 ns	31 520 540 ns	112 787 278 ns

Table 3.1: The runtimes of the second convolutional layer on Arm and x86, with and without SIMD.

	Arm		x86	
	Forward	Backward	Forward	Backward
Base	1 265 000 ns	1 774 000 ns	10 867 237 ns	19 407 642 ns
SIMD	545 000 ns	457 000 ns	1 787 338 ns	3 546 827 ns

Table 3.2: The runtimes of the first linear layer on Arm and x86, with and without SIMD.

References

[1] Kenjiro Taura. *parallel-distributed*. 2023. URL: `https://github.com/taura/parallel-distributed` (visited on 01/30/2023).

[2] Hendrik Kühne. *parallel-distributed*. 2023. URL: `https://github.com/HendrikKuehne/parallel-distributed` (visited on 01/30/2023).

[3] Arm Limited. *Vector data types for NEON intrinsics*. 2023. URL: `https://developer.arm.com/documentation/den0018/a/NEON-Intrinsics/Vector-data-types-for-NEON-intrinsics` (visited on 01/30/2023).

[4] Arm Limited. *Arm Intrinsics*. 2023. URL: `https://developer.arm.com/architectures/instruction-sets/intrinsics/` (visited on 01/30/2023).

[5] Intel Corporation. *Intel Intrinsics Guide*. 2022. URL: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html` (visited on 01/30/2023).