Advanced Embedded Systems

Answers for Assignment 2

SILVIA LUCIA SANNA – 70/90/00053

First step: assignment config

Before starting to code and make the ReLU functions, I have to define my configuration:

- The type of data I use is defined by looking the result of my last student id digit module with 3. So, the last digit of my student id is 3 and if I make the module operation with 3: 3%3=0 so the type of data I use is char. Making the ReLU with characters is a bit senseless so I transformed my data type to an unsigned int made with 8 bits: data type char is made with 8 bits.
- I have to use a loop unrolling of 4 (unroll by a factor 4) because the second last digit of my student id is 5 whose module with 3 is 2: 5%3=2. Making an unrolling of 4 means that I am using all 4 cores of my architecture: each one is 128 bit (defined in the architecture) and so when later on I will use NEON registers, this means that each core will accept 1 NEON register (each NEON register is 128 bit).

```
#define DIM 260;

uint8_t a[DIM];

for (int j=0; j<DIM; j++){

    a[j]= j; //fill in a with "random" numbers
}

uint8_t b[DIM];

//I compare each value with 127 (so I know if the sign bit is 0 or 1)

int threshold=127;
```

Figure 1: declaration of my variables

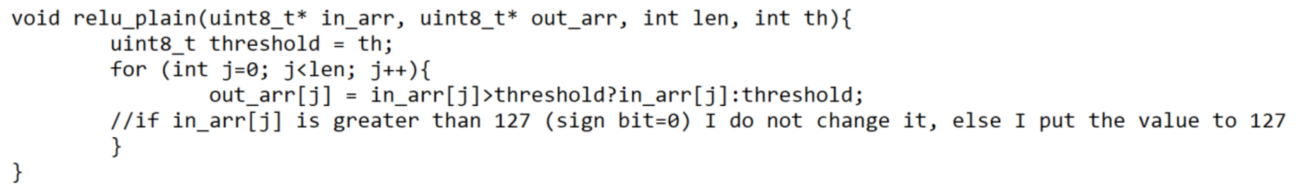
Second step: ReLU function in C

To solve this step, first I have to define what a ReLU function is. The ReLU is mostly used in neural networks and is an activation function which changes the values in a function depending on a certain threshold. Therefore, I have a threshold and look for every point of the function: if this value is less than the threshold I change this value to another one previously defined and usualy 0 but can be also the maximum value or the mean value, whatever I want and makes sense with my project; if the point is greater than the threshold, I leave the value unchanged.

Before proceeding with the ReLU function, I have to define an array in which I have to apply the ReLU and depending on my configuration scenario. As I have 4 cores each one with 128 bit and I have to use them all together, this means that my array must have at least 512 bits (4*128): as the data type of the array is uint8_t, this means that the array must have at least 64 elements (512/8), but as I also want to make more than one loop I can multiply this number for whatever I want. I have decided to make at least 4 loops, here there is no reason, I liked it. As I previously know how NEON registers and loop unrolloing works (I will explain in the next paragraph), I decided to make my vector of 260 elements.

So, the input array to give to the ReLU function, is made by 260 elements each one with 8 bits. So I create “uint8_t a[260];” and I have to fill in the array properly. With 8 bits I can express all values from 0 to 255 and I filled in with numbers from 0 to 260 (my DIM, so the length of a, I made a cycle in j form 0 to 260) but thanks to the cast array, when the j value was 256 I had like 0 value and so j=257 the value was 1 and so on up to the end.

My threshold value was 127 because it is the exact half value between 0 and 255 so it is a good way to rectify on that value, also because all values less than 127 have the significant bit set to 0 and all upper values until 255 have the significant bit set to 1.



```
void relu_plain(uint8_t* in_arr, uint8_t* out_arr, int len, int th){
        uint8_t threshold = th;
        for (int j=0; j<len; j++){
                out_arr[j] = in_arr[j]>threshold?in_arr[j]:threshold;
        //if in_arr[j] is greater than 127 (sign bit=0) I do not change it, else I put the value to 127
        }
}
```

Figure 2: Implementation of my ReLU plain function

My ReLU function was pretty general: I passed as parameters the input array, output array, length of both arrays (which is the same) and the threshold value and in the function, I make a cast with the data type used as input and output (which of course must be the same). As output array I could use also the same input array, but I preferred to not lose input values because I have to pass the same array also to the ReLU function with NEON registers, so I need a different output array.

So, in my ReLU function when the data is greater than 127 it is not changed, otherwise it is put to threshold value (127). The new value is saved in out_arr.

Third step: ReLU with NEON and loop unrolling

One of the first things to do before starting this step is to understand better what NEON and loop unrolling are and how they work.

- Neon: they are registers developed by ARM in which you can make operations with 128 bit seen as single elements so you can vectorize your computation.
- Loop unrolling is a technique that groups instructions in a single loop that would be executed in more than one loop. As you can imagine this means to make a parallelization using the different cores you have in your microprocessor. In my case I had a loop unrolling of factor 4 so this means I have to use 4 cores, each one is 128 bit so in each core I can process only 1 neon register.

So, how to structure the ReLU with NEON? My configuration was the following one: I had a vector made by 260 elements each one 8 bits and 4 neon registers. Each neon register (and so each process) can accept in my project 16 variables of the input array (128/8=16; bits_in_neon/bits_elements_inarray = #variables_in_each_neon); so in parallel I can process 64 elements of the input array simultaneously (4*16=64; #neon * elements_inarray_in1neon = #elements_in_parallel). For the total of elements in my array I have to do 4 for cycles (inf(260/16); inferior number of the division between len(a) and #elements_in_parallel) but some elements are not processed by the neon (I have an offset of 4) so I have to make a recall to relu_plain function to process all these elements in parallel.

```
void reluNEON (uint8_t* in_arr, uint8_t* out_arr, int len, int th){ //with neon registers I read 128 bit at a time
        //I have 4 cores (c0,c1,c2,c3); 1 neon block reads 128 bit,
        //each of my data is 8 bit so I have 16 neon blocks and 4 will be processed without neon
        //Each core processes 1 neon block, so 16 data each one of 8 bits;
        //I associate 4 different variables, one for each neonp
        uint8x16_t c0in, c1in, c2in, c3in; //each neon 16 variables of 8 bits each one
        uint8x16_t c0out, c1out, c2out, c3out;
        int offset = len % 64;
        uint8x16_t threshold = vdupq_n_u8(th);
        for (int j=0; j<len/64; j++, in_arr+=64, out_arr+=64){
            c0in = vld1q_u8(in_arr);
            c1in = vld1q_u8(in_arr + 16); //I have 16 variables of the input array in each neon
            c2in = vld1q_u8(in_arr + 32);
            c3in = vld1q_u8(in_arr + 48);

            c0out = vmaxq_u8(c0in, threshold);
            c1out = vmaxq_u8(c1in, threshold);
            c2out = vmaxq_u8(c2in, threshold);
            c3out = vmaxq_u8(c3in, threshold);

            vst1q_u8(out_arr, c0out);
            vst1q_u8(out_arr + 16, c1out);
            vst1q_u8(out_arr + 32, c2out);
            vst1q_u8(out_arr + 48, c3out);
        }
        //to process individually remaining elements (not processed during parallelization)
        uint8_t threshold_plain = th;
        for (int j=0; j<offset; j++){
                out_arr[j] = in_arr[j]>threshold_plain?in_arr[j]:threshold_plain; }
        //if in_arr[j] is greater than 127 (sign bit=0) I do not change it, else I put the value to 127}

}
```

Figure 3: Implementation of ReLU with Neon registers

Briefly explaining my ReLU function with neon registers, I have 6 main steps:

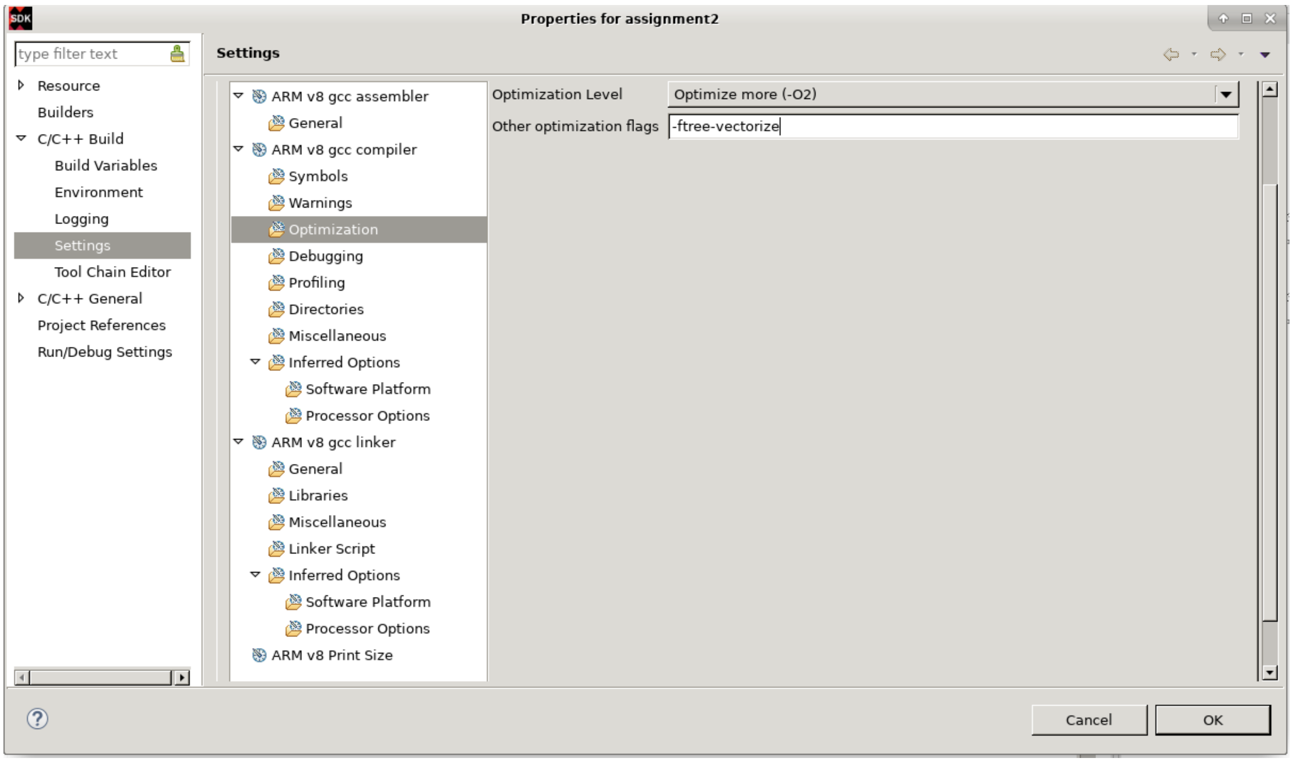
1. I create the neon registers to process input array and output array (c0in, c1in, c2in, c3in, c0out, c1out, c2out, c3out) each one of type 8x16 because they process 16 elements of 8 bits.
2. Then I define the offset (elements I have to process lonely) and make a cast in the type variable for the threshold: as input and output array are type uint8x16_t also this one must be the same type and with function vdupq_n_u8 I can have an array made by 128 bits (q register) containing different variables with 8 bits each one near to the other.

3. After that I have a loop in which I process the 64 elements previously mentioned, and this repeats 4 times. The first thing to do in this loop is to fill-in neon registers (created in point 1 of this list) appropriately, each one containing 16 elements of the input array. In fact the first one will have from element 0 to 15, the second one from 16 to 31 (in_arr+16), the third one from 32 to 47 (in_arr+32) and the last one from 48 to 63 (in_arr+48). Here the function vld1q_u8 is very important because loads the 16 elements of the input array each one with 8 bits in a register of 128 bits called q.
4. Again, in the loop another important step is to check if each element of the input array is greater or not than the threshold. This is done using the function vmaxq_u8 which checks element per element if is greater than the threshold or not and rectifies it putting the result in the appropriate output core.
5. The last thing to do in the loop is to fill in the output array with the values stored in the correct core, done thanks to function vst1q_u8 which stores elements of 128 q in the output array.
6. After that I have reimplemented the relu_plain, I did not call it in the previous implementation because I feared to have a very hard assembly in that part. Here the relu_plain is only for elements not processed during parallelization that in my case are 4.

Fourth step: ReLU plain with auto-vectorization

The auto-vectorization is a technique used by the compiler to automize the compilation, in fact in the assembly I have less useless instructions (like stores and loads for security to not lose results). I can select how auto-vectorize: usually when I compile a code using gcc I can set the parameter -o N with a number from 1 to 3 standing for the optimization level. Otherwise, as I did in this assignment, in the SDK I can make “right click on the project → click on C/C++ build settings → Tool setting → ARMv8 gcc compiler → Optimization” and here select the level and also the flag.

In this assignment I have selected level 2 for optimization (I did not want a very strong optimization and this was the intermediate one) and also filled in the flag “other optimization flags” where I put -ftree-vectorize to make an automated computation with NEON registers. For this reason, the auto-vectorization here is applied to only function relu_plain.



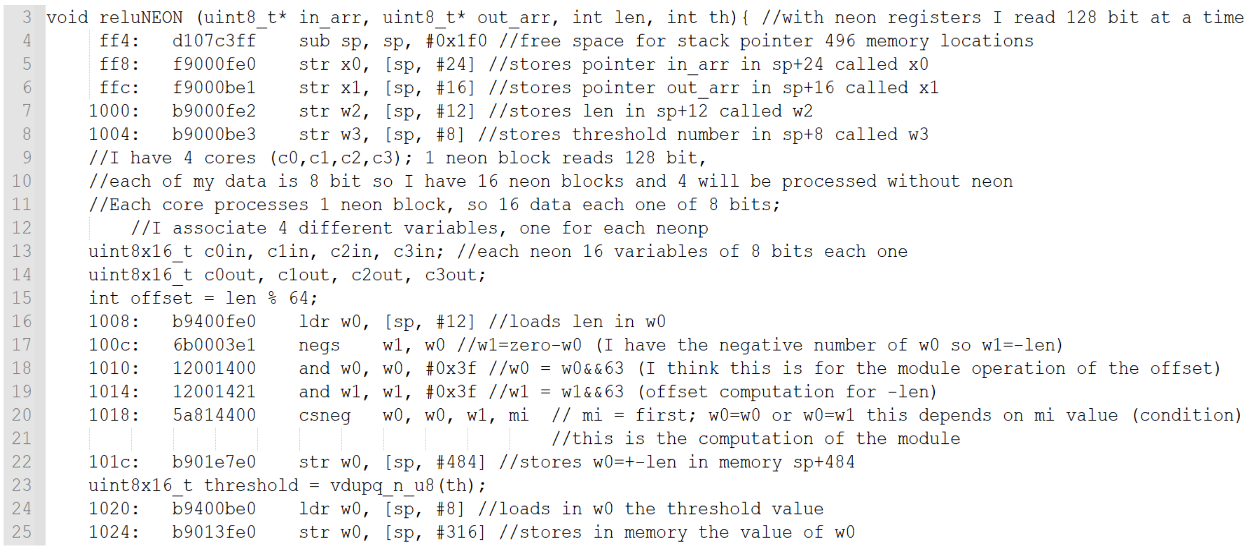
Optimization Level: Optimize more (-O2)
Other optimization flags: -ftree-vectorize

Figure 4: How to set optimization level and flag for autovectorization

Fifth step: comment assembly

The assembly I commented for this assignment is only the one related to ReLU_NEON function. You can read my assembly commented in the file called “assembly_RELUNEON.txt”. Here I briefly explain what happens. In the file I have main blocks:

1. In the first 4 lines I have the initialization of the variables that the function receives as input and also the initialization of variables like offset and threshold

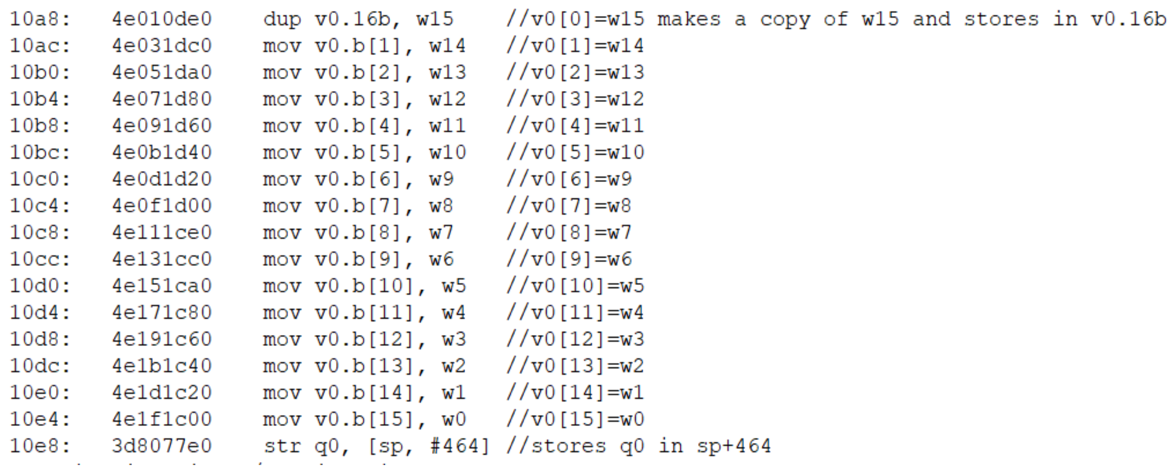


```
void reluNEON (uint8_t* in_arr, uint8_t* out_arr, int len, int th){ //with neon registers I read 128 bit at a time
     ff4:   d107c3ff    sub sp, sp, #0x1f0 //free space for stack pointer 496 memory locations
     ff8:   f9000fe0    str x0, [sp, #24] //stores pointer in_arr in sp+24 called x0
     ffc:   f9000be1    str x1, [sp, #16] //stores pointer out_arr in sp+16 called x1
    1000:   b9000fe2    str w2, [sp, #12] //stores len in sp+12 called w2
    1004:   b9000be3    str w3, [sp, #8] //stores threshold number in sp+8 called w3
    //I have 4 cores (c0,c1,c2,c3); 1 neon block reads 128 bit,
    //each of my data is 8 bit so I have 16 neon blocks and 4 will be processed without neon
    //Each core processes 1 neon block, so 16 data each one of 8 bits;
        //I associate 4 different variables, one for each neonp
    uint8x16_t c0in, c1in, c2in, c3in; //each neon 16 variables of 8 bits each one
    uint8x16_t c0out, c1out, c2out, c3out;
    int offset = len % 64;
    1008:   b9400fe0    ldr w0, [sp, #12] //loads len in w0
    100c:   6b0003e1    negs    w1, w0 //w1=zero-w0 (I have the negative number of w0 so w1=-len)
    1010:   12001400    and w0, w0, #0x3f //w0 = w0&&63 (I think this is for the module operation of the offset)
    1014:   12001421    and w1, w1, #0x3f //w1 = w1&&63 (offset computation for -len)
    1018:   5a814400    csneg   w0, w0, w1, mi  // mi = first; w0=w0 or w0=w1 this depends on mi value (condition)
                                                //this is the computation of the module
    101c:   b901e7e0    str w0, [sp, #484] //stores w0=+-len in memory sp+484
    uint8x16_t threshold = vdupq_n_u8(th);
    1020:   b9400be0    ldr w0, [sp, #8] //loads in w0 the threshold value
    1024:   b9013fe0    str w0, [sp, #316] //stores in memory the value of w0
```

Figure 5: first part of my assembly, variable initialization

2. The second important block is the store of the threshold variable in a register q of 128 bits

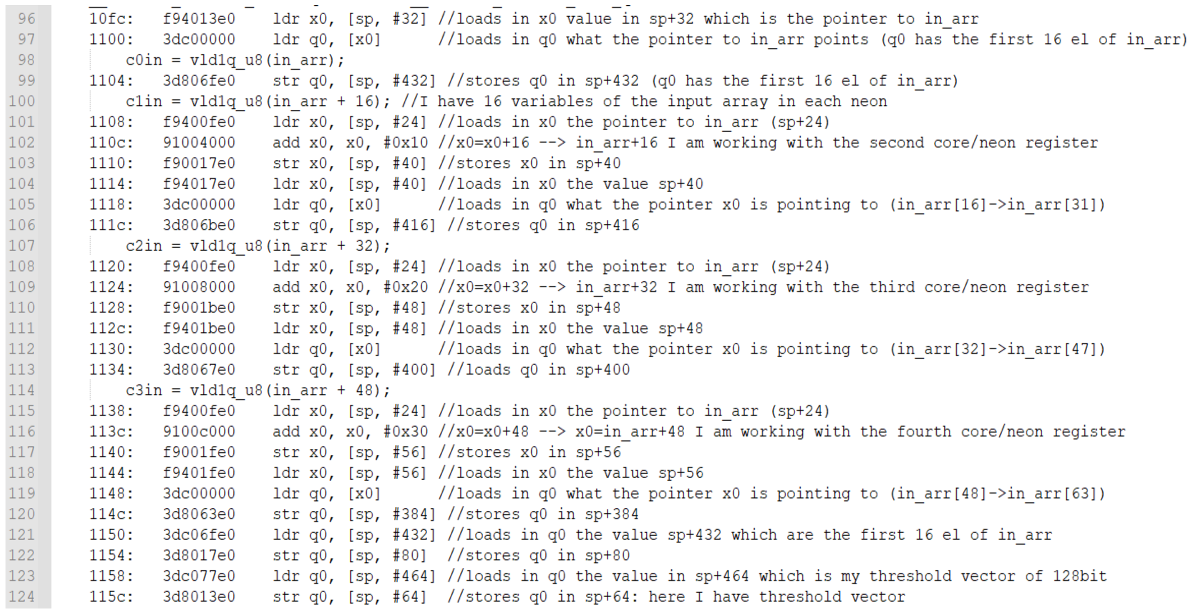
```
1028:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
102c:   12001c0f    and w15, w0, #0xff //makes a mask between 255 (the first 8 bits) and w0 storing in w15
1030:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1034:   12001c0e    and w14, w0, #0xff //makes a mask between 255 (the first 8 bits) and w0 storing in w14
1038:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
103c:   12001c0d    and w13, w0, #0xff //makes a mask between 255 (the first 8 bits) and w0 storing in w13
1040:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1044:   12001c0c    and w12, w0, #0xff //makes a mask between 255 (the first 8 bits) and w0 storing in w12
1048:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
104c:   12001c0b    and w11, w0, #0xff //makes a mask between 255 (the first 8 bits) and w0 storing in w11
1050:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1054:   12001c0a    and w10, w0, #0xff //makes a mask between 255 (the first 8 bits) and w0 storing in w10
1058:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
105c:   12001c09    and w9, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w9
1060:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1064:   12001c08    and w8, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w8
1068:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
106c:   12001c07    and w7, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w7
1070:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1074:   12001c06    and w6, w0, #0xff   //makes a mask between 255 (the first 8 bits) and w0 storing in w6
1078:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
107c:   12001c05    and w5, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w5
1080:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1084:   12001c04    and w4, w0, #0xff   //makes a mask between 255 (the first 8 bits) and w0 storing in w4
1088:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
108c:   12001c03    and w3, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w3
1090:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
1094:   12001c02    and w2, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w2
1098:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
109c:   12001c01    and w1, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w1
10a0:   b9413fe0    ldr w0, [sp, #316] //loads in w0 the threshold value
10a4:   12001c00    and w0, w0, #0xff  //makes a mask between 255 (the first 8 bits) and w0 storing in w0
```



```
10a8:   4e010de0    dup v0.16b, w15    //v0[0]=w15 makes a copy of w15 and stores in v0.16b
10ac:   4e031dc0    mov v0.b[1], w14   //v0[1]=w14
10b0:   4e051da0    mov v0.b[2], w13   //v0[2]=w13
10b4:   4e071d80    mov v0.b[3], w12   //v0[3]=w12
10b8:   4e091d60    mov v0.b[4], w11   //v0[4]=w11
10bc:   4e0b1d40    mov v0.b[5], w10   //v0[5]=w10
10c0:   4e0d1d20    mov v0.b[6], w9    //v0[6]=w9
10c4:   4e0f1d00    mov v0.b[7], w8    //v0[7]=w8
10c8:   4e111ce0    mov v0.b[8], w7    //v0[8]=w7
10cc:   4e131cc0    mov v0.b[9], w6    //v0[9]=w6
10d0:   4e151ca0    mov v0.b[10], w5   //v0[10]=w5
10d4:   4e171c80    mov v0.b[11], w4   //v0[11]=w4
10d8:   4e191c60    mov v0.b[12], w3   //v0[12]=w3
10dc:   4e1b1c40    mov v0.b[13], w2   //v0[13]=w2
10e0:   4e1d1c20    mov v0.b[14], w1   //v0[14]=w1
10e4:   4e1f1c00    mov v0.b[15], w0   //v0[15]=w0
10e8:   3d8077e0    str q0, [sp, #464] //stores q0 in sp+464
```

Figure 6: stores threshold value in a q register

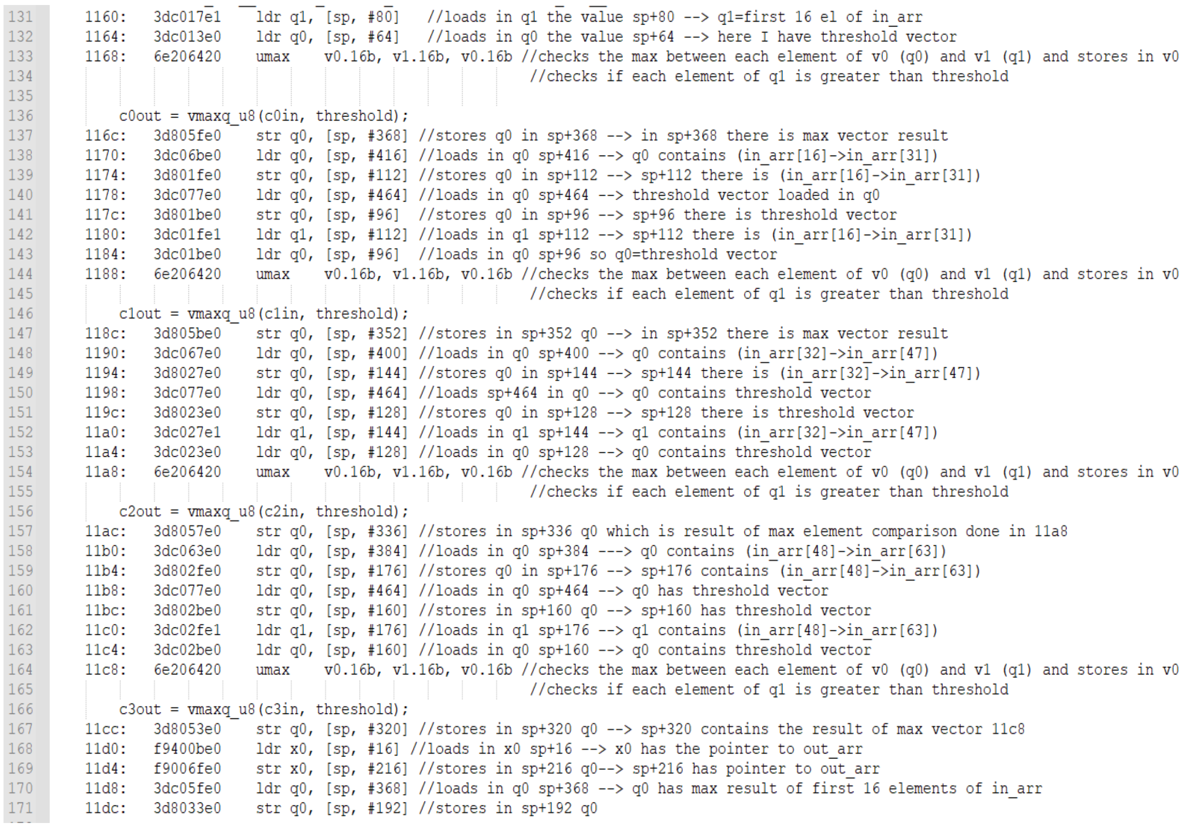
3. The other important block is the load in neon registers of the input array values



```
10fc:   f94013e0    ldr x0, [sp, #32] //loads in x0 value in sp+32 which is the pointer to in_arr
1100:   3dc00000    ldr q0, [x0]      //loads in q0 what the pointer to in_arr points (q0 has the first 16 el of in_arr)
    c0in = vld1q_u8(in_arr);
1104:   3d806fe0    str q0, [sp, #432] //stores q0 in sp+432 (q0 has the first 16 el of in_arr)
    c1in = vld1q_u8(in_arr + 16); //I have 16 variables of the input array in each neon
1108:   f9400fe0    ldr x0, [sp, #24] //loads in x0 the pointer to in_arr (sp+24)
110c:   91004000    add x0, x0, #0x10 //x0=x0+16 --> in_arr+16 I am working with the second core/neon register
1110:   f90017e0    str x0, [sp, #40] //stores x0 in sp+40
1114:   f94017e0    ldr x0, [sp, #40] //loads in x0 the value sp+40
1118:   3dc00000    ldr q0, [x0]      //loads in q0 what the pointer x0 is pointing to (in_arr[16]->in_arr[31])
111c:   3d806be0    str q0, [sp, #416] //stores q0 in sp+416
    c2in = vld1q_u8(in_arr + 32);
1120:   f9400fe0    ldr x0, [sp, #24] //loads in x0 the pointer to in_arr (sp+24)
1124:   91008000    add x0, x0, #0x20 //x0=x0+32 --> in_arr+32 I am working with the third core/neon register
1128:   f9001be0    str x0, [sp, #48] //stores x0 in sp+48
112c:   f9401be0    ldr x0, [sp, #48] //loads in x0 the value sp+48
1130:   3dc00000    ldr q0, [x0]      //loads in q0 what the pointer x0 is pointing to (in_arr[32]->in_arr[47])
1134:   3d8067e0    str q0, [sp, #400] //loads q0 in sp+400
    c3in = vld1q_u8(in_arr + 48);
1138:   f9400fe0    ldr x0, [sp, #24] //loads in x0 the pointer to in_arr (sp+24)
113c:   9100c000    add x0, x0, #0x30 //x0=x0+48 --> x0=in_arr+48 I am working with the fourth core/neon register
1140:   f9001fe0    str x0, [sp, #56] //stores x0 in sp+56
1144:   f9401fe0    ldr x0, [sp, #56] //loads in x0 the value sp+56
1148:   3dc00000    ldr q0, [x0]      //loads in q0 what the pointer x0 is pointing to (in_arr[48]->in_arr[63])
114c:   3d8063e0    str q0, [sp, #384] //stores q0 in sp+384
1150:   3dc06fe0    ldr q0, [sp, #432] //loads in q0 the value sp+432 which are the first 16 el of in_arr
1154:   3d8017e0    str q0, [sp, #80]  //stores q0 in sp+80
1158:   3dc077e0    ldr q0, [sp, #464] //loads in q0 the value in sp+464 which is my threshold vector of 128bit
115c:   3d8013e0    str q0, [sp, #64]  //stores q0 in sp+64: here I have threshold vector
```

Figure 7: load part in the assembly

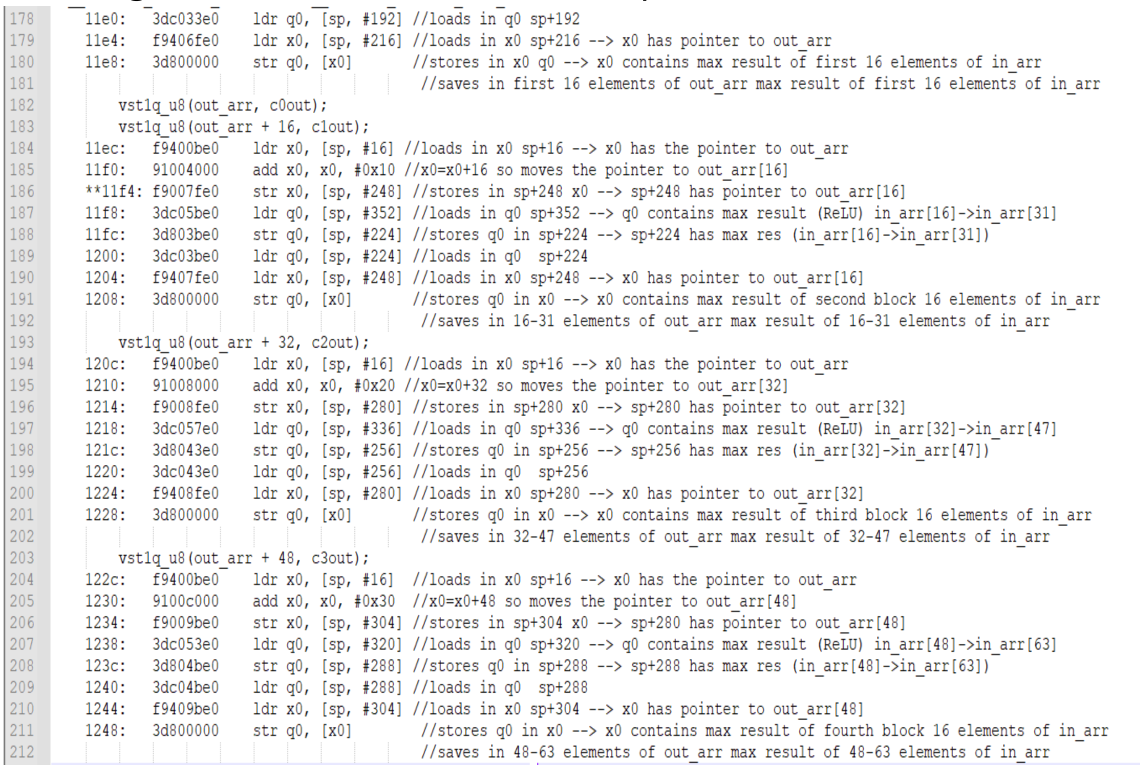
4. As in the C code after this there is the check for each value of the input array to rectify it or not with respect to the relation with threshold value



```
   1160:   3dc017e1    ldr q1, [sp, #80]   //loads in q1 the value sp+80 --> q1=first 16 el of in_arr
   1164:   3dc013e0    ldr q0, [sp, #64]   //loads in q0 the value sp+64 --> here I have threshold vector
   1168:   6e206420    umax    v0.16b, v1.16b, v0.16b //checks the max between each element of v0 (q0) and v1 (q1) and stores in v0
                                                      //checks if each element of q1 is greater than threshold

       c0out = vmaxq_u8(c0in, threshold);
   116c:   3d805fe0    str q0, [sp, #368] //stores q0 in sp+368 --> in sp+368 there is max vector result
   1170:   3dc06be0    ldr q0, [sp, #416] //loads in q0 sp+416 --> q0 contains (in_arr[16]->in_arr[31])
   1174:   3d801fe0    str q0, [sp, #112] //stores q0 in sp+112 --> sp+112 there is (in_arr[16]->in_arr[31])
   1178:   3dc077e0    ldr q0, [sp, #464] //loads in q0 sp+464 --> threshold vector loaded in q0
   117c:   3d801be0    str q0, [sp, #96]  //stores q0 in sp+96 --> sp+96 there is threshold vector
   1180:   3dc01fe1    ldr q1, [sp, #112] //loads in q1 sp+112 --> sp+112 there is (in_arr[16]->in_arr[31])
   1184:   3dc01be0    ldr q0, [sp, #96]  //loads in q0 sp+96 so q0=threshold vector
   1188:   6e206420    umax    v0.16b, v1.16b, v0.16b //checks the max between each element of v0 (q0) and v1 (q1) and stores in v0
                                                      //checks if each element of q1 is greater than threshold
       c1out = vmaxq_u8(c1in, threshold);
   118c:   3d805be0    str q0, [sp, #352] //stores in sp+352 q0 --> in sp+352 there is max vector result
   1190:   3dc067e0    ldr q0, [sp, #400] //loads in q0 sp+400 --> q0 contains (in_arr[32]->in_arr[47])
   1194:   3d8027e0    str q0, [sp, #144] //stores q0 in sp+144 --> sp+144 there is (in_arr[32]->in_arr[47])
   1198:   3dc077e0    ldr q0, [sp, #464] //loads sp+464 in q0 --> q0 contains threshold vector
   119c:   3d8023e0    str q0, [sp, #128] //stores q0 in sp+128 --> sp+128 there is threshold vector
   11a0:   3dc027e1    ldr q1, [sp, #144] //loads in q1 sp+144 --> q1 contains (in_arr[32]->in_arr[47])
   11a4:   3dc023e0    ldr q0, [sp, #128] //loads in q0 sp+128 --> q0 contains threshold vector
   11a8:   6e206420    umax    v0.16b, v1.16b, v0.16b //checks the max between each element of v0 (q0) and v1 (q1) and stores in v0
                                                      //checks if each element of q1 is greater than threshold
       c2out = vmaxq_u8(c2in, threshold);
   11ac:   3d8057e0    str q0, [sp, #336] //stores in sp+336 q0 which is result of max element comparison done in 11a8
   11b0:   3dc063e0    ldr q0, [sp, #384] //loads in q0 sp+384 ---> q0 contains (in_arr[48]->in_arr[63])
   11b4:   3d802fe0    str q0, [sp, #176] //stores in sp+176 --> sp+176 contains (in_arr[48]->in_arr[63])
   11b8:   3dc077e0    ldr q0, [sp, #464] //loads in q0 sp+464 --> q0 has threshold vector
   11bc:   3d802be0    str q0, [sp, #160] //stores in sp+160 q0 --> sp+160 has threshold vector
   11c0:   3dc02fe1    ldr q1, [sp, #176] //loads in q1 sp+176 --> q1 contains (in_arr[48]->in_arr[63])
   11c4:   3dc02be0    ldr q0, [sp, #160] //loads in q0 sp+160 --> q0 contains threshold vector
   11c8:   6e206420    umax    v0.16b, v1.16b, v0.16b //checks the max between each element of v0 (q0) and v1 (q1) and stores in v0
                                                      //checks if each element of q1 is greater than threshold
       c3out = vmaxq_u8(c3in, threshold);
   11cc:   3d8053e0    str q0, [sp, #320] //stores in sp+320 q0 --> sp+320 contains the result of max vector 11c8
   11d0:   f9400be0    ldr x0, [sp, #16] //loads in x0 sp+16 --> x0 has the pointer to out_arr
   11d4:   f9006fe0    str x0, [sp, #216] //stores in sp+216 q0--> sp+216 has pointer to out_arr
   11d8:   3dc05fe0    ldr q0, [sp, #368] //loads in q0 sp+368 --> q0 has max result of first 16 elements of in_arr
   11dc:   3d8033e0    str q0, [sp, #192] //stores in sp+192 q0
```

Figure 8: ReLU computation in assembly (check max value pair by pair between input array and threshold array)

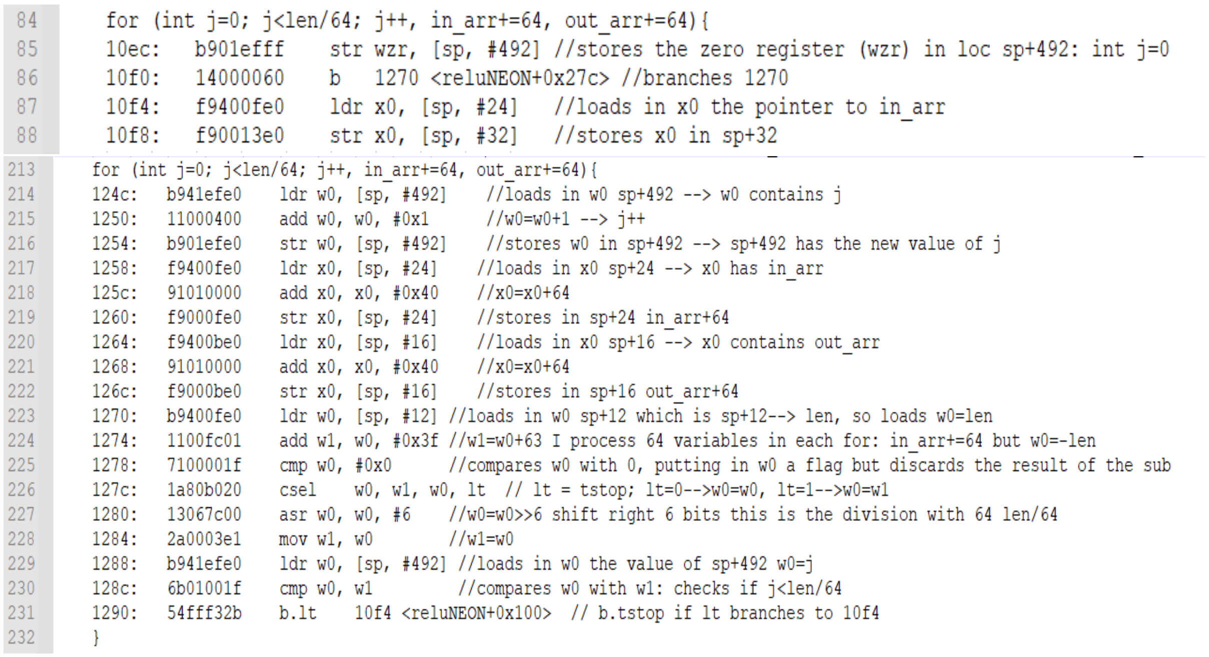
5. As the ReLU results are computed, we have to store them in the registers that I use as output



```
   11e0:   3dc033e0    ldr q0, [sp, #192] //loads in q0 sp+192
   11e4:   f9406fe0    ldr x0, [sp, #216] //loads in x0 sp+216 --> x0 has pointer to out_arr
   11e8:   3d800000    str q0, [x0]       //stores in x0 q0 --> x0 contains max result of first 16 elements of in_arr
                                           //saves in first 16 elements of out_arr max result of first 16 elements of in_arr
       vst1q_u8(out_arr, c0out);
       vst1q_u8(out_arr + 16, c1out);
   11ec:   f9400be0    ldr x0, [sp, #16] //loads in x0 sp+16 --> x0 has the pointer to out_arr
   11f0:   91004000    add x0, x0, #0x10 //x0=x0+16 so moves the pointer to out_arr[16]
   **11f4: f9007fe0    str x0, [sp, #248] //stores in sp+248 x0 --> sp+248 has pointer to out_arr[16]
   11f8:   3dc05be0    ldr q0, [sp, #352] //loads in q0 sp+352 --> q0 contains max result (ReLU) in_arr[16]->in_arr[31]
   11fc:   3d803be0    str q0, [sp, #224] //stores q0 in sp+224 --> sp+224 has max res (in_arr[16]->in_arr[31])
   1200:   3dc03be0    ldr q0, [sp, #224] //loads in q0  sp+224
   1204:   f9407fe0    ldr x0, [sp, #248] //loads in x0 sp+248 --> x0 has pointer to out_arr[16]
   1208:   3d800000    str q0, [x0]       //stores q0 in x0 --> x0 contains max result of second block 16 elements of in_arr
                                           //saves in 16-31 elements of out_arr max result of 16-31 elements of in_arr
       vst1q_u8(out_arr + 32, c2out);
   120c:   f9400be0    ldr x0, [sp, #16] //loads in x0 sp+16 --> x0 has the pointer to out_arr
   1210:   91008000    add x0, x0, #0x20 //x0=x0+32 so moves the pointer to out_arr[32]
   1214:   f9008fe0    str x0, [sp, #280] //stores in sp+280 x0 --> sp+280 has pointer to out_arr[32]
   1218:   3dc057e0    ldr q0, [sp, #336] //loads in q0 sp+336 --> q0 contains max result (ReLU) in_arr[32]->in_arr[47]
   121c:   3d8043e0    str q0, [sp, #256] //stores q0 in sp+256 --> sp+256 has max res (in_arr[32]->in_arr[47])
   1220:   3dc043e0    ldr q0, [sp, #256] //loads in q0  sp+256
   1224:   f9408fe0    ldr x0, [sp, #280] //loads in x0 sp+280 --> x0 has pointer to out_arr[32]
   1228:   3d800000    str q0, [x0]       //stores q0 in x0 --> x0 contains max result of third block 16 elements of in_arr
                                           //saves in 32-47 elements of out_arr max result of 32-47 elements of in_arr
       vst1q_u8(out_arr + 48, c3out);
   122c:   f9400be0    ldr x0, [sp, #16]  //loads in x0 sp+16 --> x0 has the pointer to out_arr
   1230:   9100c000    add x0, x0, #0x30  //x0=x0+48 so moves the pointer to out_arr[48]
   1234:   f9009be0    str x0, [sp, #304] //stores in sp+304 x0 --> sp+280 has pointer to out_arr[48]
   1238:   3dc053e0    ldr q0, [sp, #320] //loads in q0 sp+320 --> q0 contains max result (ReLU) in_arr[48]->in_arr[63]
   123c:   3d804be0    str q0, [sp, #288] //stores q0 in sp+288 --> sp+288 has max res (in_arr[48]->in_arr[63])
   1240:   3dc04be0    ldr q0, [sp, #288] //loads in q0  sp+288
   1244:   f9409be0    ldr x0, [sp, #304] //loads in x0 sp+304 --> x0 has pointer to out_arr[48]
   1248:   3d800000    str q0, [x0]        //stores q0 in x0 --> x0 contains max result of fourth block 16 elements of in_arr
                                            //saves in 48-63 elements of out_arr max result of 48-63 elements of in_arr
```

Figure 9: store results in output

6. As I have a lot of different variables to be processed in different loop cycles, the for loop is very important and as always in assembly it is divided in two parts

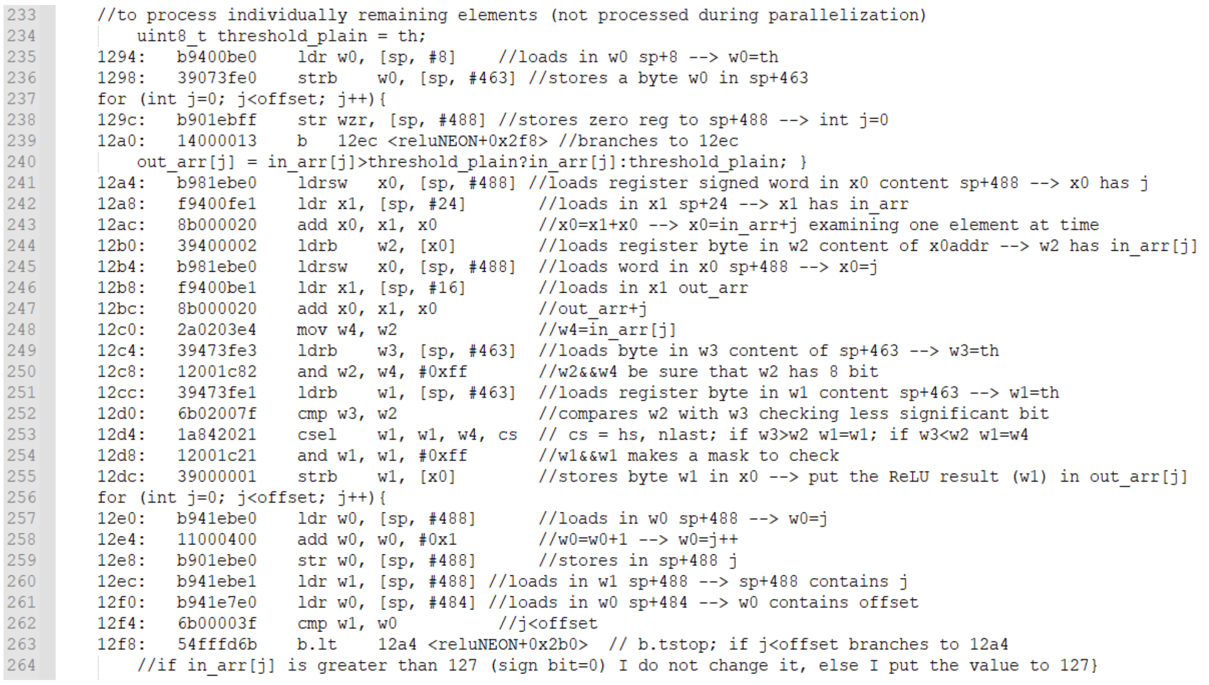


```
for (int j=0; j<len/64; j++, in_arr+=64, out_arr+=64){
10ec:   b901efff    str wzr, [sp, #492] //stores the zero register (wzr) in loc sp+492: int j=0
10f0:   14000060    b   1270 <reluNEON+0x27c> //branches 1270
10f4:   f9400fe0    ldr x0, [sp, #24]   //loads in x0 the pointer to in_arr
10f8:   f90013e0    str x0, [sp, #32]   //stores x0 in sp+32
```

```
for (int j=0; j<len/64; j++, in_arr+=64, out_arr+=64){
124c:   b941efe0    ldr w0, [sp, #492]    //loads in w0 sp+492 --> w0 contains j
1250:   11000400    add w0, w0, #0x1      //w0=w0+1 --> j++
1254:   b901efe0    str w0, [sp, #492]    //stores w0 in sp+492 --> sp+492 has the new value of j
1258:   f9400fe0    ldr x0, [sp, #24]    //loads in x0 sp+24 --> x0 has in_arr
125c:   91010000    add x0, x0, #0x40    //x0=x0+64
1260:   f9000fe0    str x0, [sp, #24]    //stores in sp+24 in_arr+64
1264:   f9400be0    ldr x0, [sp, #16]    //loads in x0 sp+16 --> x0 contains out_arr
1268:   91010000    add x0, x0, #0x40    //x0=x0+64
126c:   f9000be0    str x0, [sp, #16]    //stores in sp+16 out_arr+64
1270:   b9400fe0    ldr w0, [sp, #12] //loads in w0 sp+12 which is sp+12--> len, so loads w0=len
1274:   1100fc01    add w1, w0, #0x3f //w1=w0+63 I process 64 variables in each for: in_arr+=64 but w0=-len
1278:   7100001f    cmp w0, #0x0      //compares w0 with 0, putting in w0 a flag but discards the result of the sub
127c:   1a80b020    csel    w0, w1, w0, lt  // lt = tstop; lt=0-->w0=w0, lt=1-->w0=w1
1280:   13067c00    asr w0, w0, #6    //w0=w0>>6 shift right 6 bits this is the division with 64 len/64
1284:   2a0003e1    mov w1, w0        //w1=w0
1288:   b941efe0    ldr w0, [sp, #492] //loads in w0 the value of sp+492 w0=j
128c:   6b01001f    cmp w0, w1         //compares w0 with w1: checks if j<len/64
1290:   54fff32b    b.lt    10f4 <reluNEON+0x100>  // b.tstop if lt branches to 10f4
}
```

Figure 10: for loop assembly

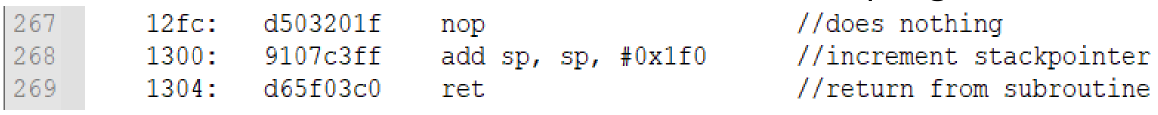
7. After this I have to compute the ReLU for the elements not processed during parallelization which is the assembly for relu_plain function



```
//to process individually remaining elements (not processed during parallelization)
    uint8_t threshold_plain = th;
1294:   b9400be0    ldr w0, [sp, #8]    //loads in w0 sp+8 --> w0=th
1298:   39073fe0    strb    w0, [sp, #463] //stores a byte w0 in sp+463
for (int j=0; j<offset; j++){
129c:   b901ebff    str wzr, [sp, #488] //stores zero reg to sp+488 --> int j=0
12a0:   14000013    b   12ec <reluNEON+0x2f8> //branches to 12ec
    out_arr[j] = in_arr[j]>threshold_plain?in_arr[j]:threshold_plain; }
12a4:   b981ebe0    ldrsw   x0, [sp, #488] //loads register signed word in x0 content sp+488 --> x0 has j
12a8:   f9400fe1    ldr x1, [sp, #24]       //loads in x1 sp+24 --> x1 has in_arr
12ac:   8b000020    add x0, x1, x0          //x0=x1+x0 --> x0=in_arr+j examining one element at time
12b0:   39400002    ldrb    w2, [x0]        //loads register byte in w2 content of x0addr --> w2 has in_arr[j]
12b4:   b981ebe0    ldrsw   x0, [sp, #488]  //loads word in x0 sp+488 --> x0=j
12b8:   f9400be1    ldr x1, [sp, #16]       //loads in x1 out_arr
12bc:   8b000020    add x0, x1, x0          //out_arr+j
12c0:   2a0203e4    mov w4, w2              //w4=in_arr[j]
12c4:   39473fe3    ldrb    w3, [sp, #463]  //loads byte in w3 content of sp+463 --> w3=th
12c8:   12001c82    and w2, w4, #0xff       //w2&&w4 be sure that w2 has 8 bit
12cc:   39473fe1    ldrb    w1, [sp, #463]  //loads register byte in w1 content sp+463 --> w1=th
12d0:   6b02007f    cmp w3, w2              //compares w2 with w3 checking less significant bit
12d4:   1a842021    csel    w1, w1, w4, cs  // cs = hs, nlast; if w3>w2 w1=w1; if w3<w2 w1=w4
12d8:   12001c21    and w1, w1, #0xff       //w1&&w1 makes a mask to check
12dc:   39000001    strb    w1, [x0]        //stores byte w1 in x0 --> put the ReLU result (w1) in out_arr[j]
for (int j=0; j<offset; j++){
12e0:   b941ebe0    ldr w0, [sp, #488]      //loads in w0 sp+488 --> w0=j
12e4:   11000400    add w0, w0, #0x1        //w0=w0+1 --> w0=j++
12e8:   b901ebe0    str w0, [sp, #488]      //stores in sp+488 j
12ec:   b941ebe1    ldr w1, [sp, #488] //loads in w1 sp+488 --> sp+488 contains j
12f0:   b941e7e0    ldr w0, [sp, #484] //loads in w0 sp+484 --> w0 contains offset
12f4:   6b00003f    cmp w1, w0          //j<offset
12f8:   54fffd6b    b.lt    12a4 <reluNEON+0x2b0>  // b.tstop; if j<offset branches to 12a4
    //if in_arr[j] is greater than 127 (sign bit=0) I do not change it, else I put the value to 127}
```

Figure 11: relu_plain function in assembly

8. The last part is the come back of the stack pointer and the return from subroutine, so the end of the program.



```
12fc:   d503201f    nop                         //does nothing
1300:   9107c3ff    add sp, sp, #0x1f0          //increment stackpointer
1304:   d65f03c0    ret                         //return from subroutine
```

Figure 12: end of the function

In all my assembly I have found these main instructions:

- Arithmetic operations:
    - sub: expresses the subtraction between two registers (the two most right) and stores the result in the first register you find in the expression. I have found this expression in my assembly only in the initial lines when I need to free space in the memory to accept the function, so to change value of the stack pointer.
    - Add: expresses the addition between two registers (the second one and the third one which, this one, can also be a number) and stores the result in the first register.
    - negs: negates the value of the register, so makes a subtraction between register zero and the second one expressed in the instruction, storing the result in the first register.
    - and: makes the logical and operation between the value of two registers, storing the result in the first register. I have found this in my assembly only when I made a mask to check the correct number of bits in a register.
    - csneg: is the conditional select negation that makes a negation of the register only if the condition (flag) expressed in the last term of the instruction is true. I have found this only once in my assembly when there was the computation for the module operation.
    - umax: checks the maximum unsigned value between a pair of elements corresponding to two different arrays. I have found this in the function vmaxq_u8 when checking if each element of the input array was greater or not than the threshold value.

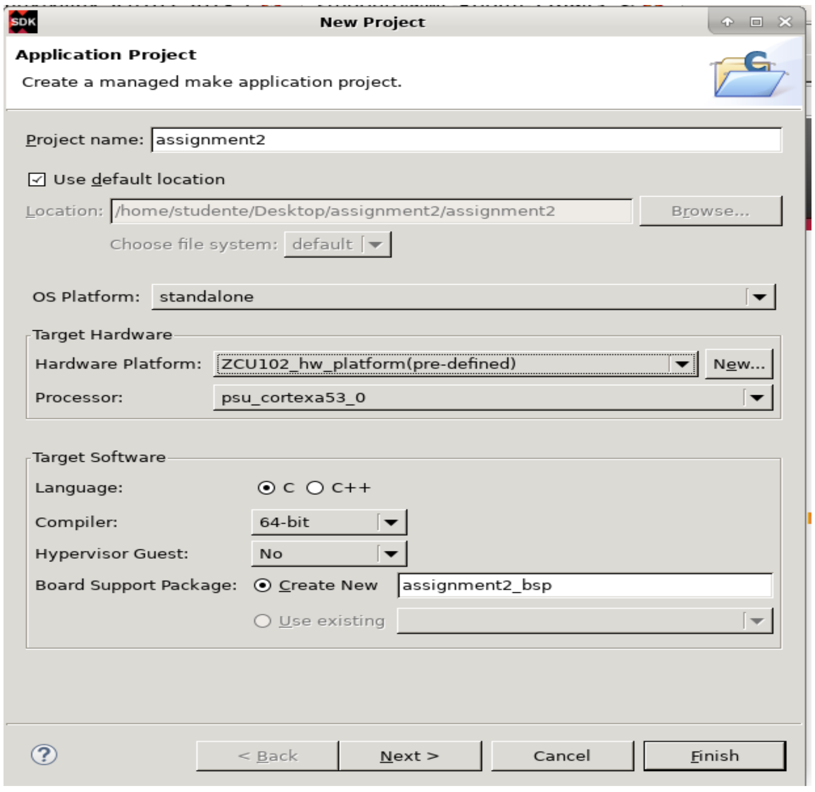
  - cmp: makes a comparison between two different registers and stores the condition in the following instruction. I have found instructions with flag lt expressing less than and with flag cs that stands for carry set (unsigned higher).
  - csel: returns the first or the second input depending on the condition found at the end of the instruction. In my assembly these instructions are with conditions lt and cs explained in the previous point.
  - asr: makes an arithmetic right shift which in assembly corresponds to make a division in fact I found this in the for loop when I have to check the end of the loop (j<len/64).
  - nop: is the instruction to not make operations, usually after an instruction that needs some delay. I have found this on my assembly at the end of the code.
  - dup: duplicates the content of a register into a scalar value. I find this only when I make the store of the 16 threshold registers from 32 bits to a vector register of 16 elements.
- Store and load operations:
  - str: instruction to store the value of a register in a certain address of the memory.
  - strb: stores a register byte in a certain memory area. I find this in my assembly only in the part of the relu_plain function when I work with single elements of the input array not processed in parallel.
  - ldr: loads the content of a certain address memory into a specific register.
  - ldrsw: loads register signed word. I find this in my assembly only in relu_plain part related to j value
  - ldrb: loads a register byte, here again I find only in relu_plain part when processing single elements of input array not processed during parallelization.
  - mov: moves the content of a register into another one, I have found this instruction only when there is the copy from register w into register v.16b with threshold content.

- Branch operations:
  - b is the instruction to branch to another instruction, used in loop cycles.
  - b.lt is the instruction to come back to the start of the loop only if the condition is satisfied (I have found only with lt condition).
- ret is the instruction to return from subroutine and so the end of the program before making the come back of the stack pointer with instruction add (at the start there was the subtraction of sp).

I found a characteristic on my assembly: as it was done without any optimization, there were a lot of useless instruction to store and load registers to be used in the next instruction. As there is no optimization this is done by the compiler to store values and not lose them.

Fifth step: results

First thing to know before compiling is to set the QEMU compiler and so configuring the platform in SDK settings in the proper way


New Project

Application Project

Create a managed make application project.

Project name: assignment2

☑ Use default location

Location: /home/studente/Desktop/assignment2/assignment2

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: ZCU102_hw_platform(pre-defined)

Processor: psu_cortexa53_0

Target Software

Language: [✓ C] C++

Compiler: 64-bit

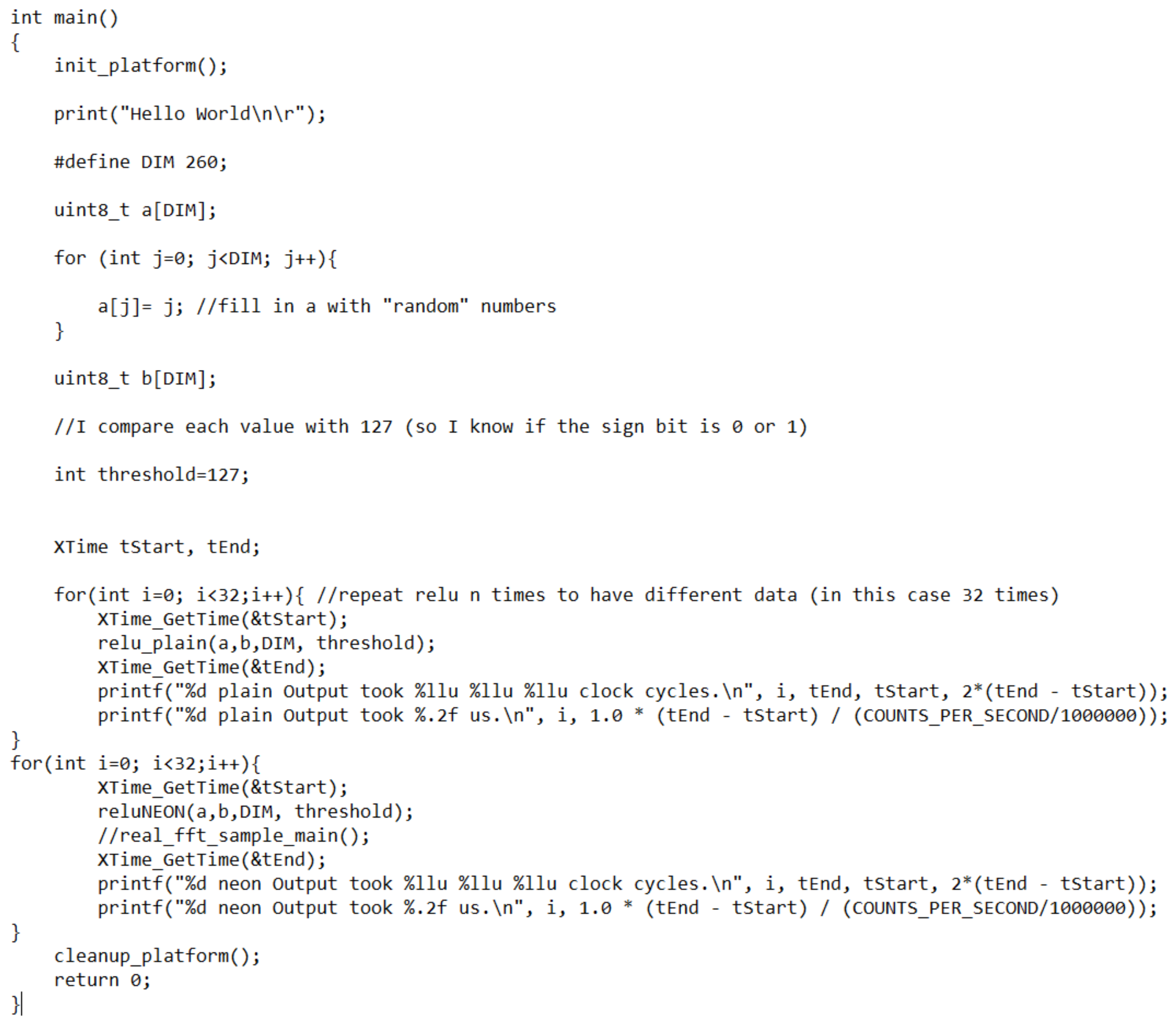
Hypervisor Guest: No

Board Support Package: [✓ Create New] assignment2_bsp

Use existing

Figure 13: set compiler

As you can see from my main function, to check the results I called the relu_plain and relu_NEON functions 32 times (it was a great number to have mean results) and compute the execution time thanks to function XTime_GetTime at the start and stop of each call to the functions.



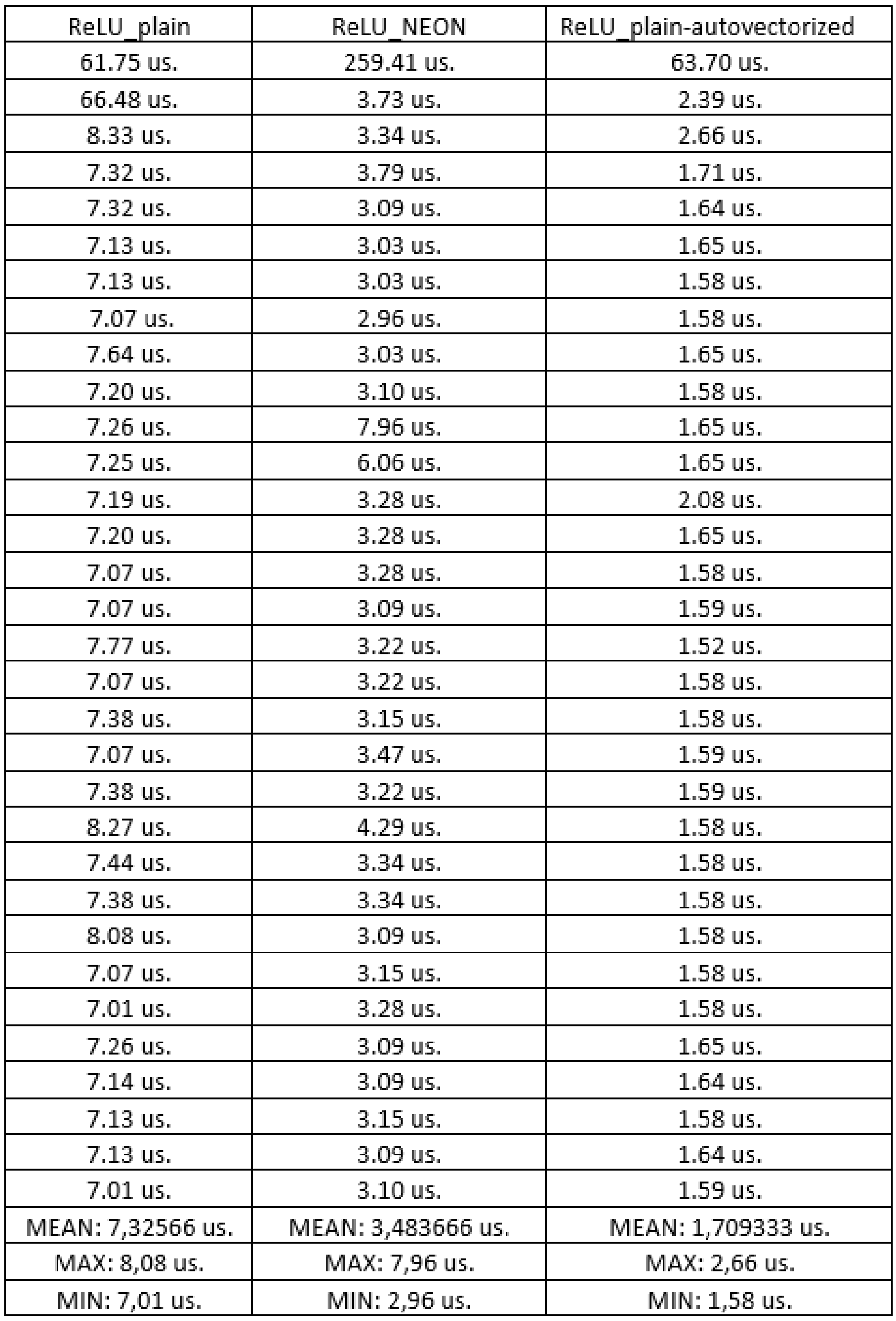
```
int main()
{
    init_platform();

    print("Hello World\n\r");

    #define DIM 260;

    uint8_t a[DIM];

    for (int j=0; j<DIM; j++){

        a[j]= j; //fill in a with "random" numbers
    }

    uint8_t b[DIM];

    //I compare each value with 127 (so I know if the sign bit is 0 or 1)

    int threshold=127;


    XTime tStart, tEnd;

    for(int i=0; i<32;i++){ //repeat relu n times to have different data (in this case 32 times)
        XTime_GetTime(&tStart);
        relu_plain(a,b,DIM, threshold);
        XTime_GetTime(&tEnd);
        printf("%d plain Output took %llu %llu %llu clock cycles.\n", i, tEnd, tStart, 2*(tEnd - tStart));
        printf("%d plain Output took %.2f us.\n", i, 1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));
}
for(int i=0; i<32;i++){
        XTime_GetTime(&tStart);
        reluNEON(a,b,DIM, threshold);
        //real_fft_sample_main();
        XTime_GetTime(&tEnd);
        printf("%d neon Output took %llu %llu %llu clock cycles.\n", i, tEnd, tStart, 2*(tEnd - tStart));
        printf("%d neon Output took %.2f us.\n", i, 1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));
}
    cleanup_platform();
    return 0;
}
```

Figure 14: main function in C

With TStart and TEnd I only have half clock cycle, so I have to make the difference between them and multiply by 2 to know the time in terms of clock cycles. To know the time in terms of microseconds I have to make a simple conversion with this formula: 1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000)

In the table below you can see the results for each iteration (from 1 to 32) for ReLU_plain without optimization (first column), for ReLU_NEON without optimization and made by me (second column) and ReLU_plain-autovectorized automatically optimized (third column).

Table 1: execution times for last 30 iterations in relu_plain, reluNEON and relu_plain autovectorized. Last three rows stands for mean, max and min values



ReLU_plain	ReLU_NEON	ReLU_plain-autovectorized
61.75 us.	259.41 us.	63.70 us.
66.48 us.	3.73 us.	2.39 us.
8.33 us.	3.34 us.	2.66 us.
7.32 us.	3.79 us.	1.71 us.
7.32 us.	3.09 us.	1.64 us.
7.13 us.	3.03 us.	1.65 us.
7.13 us.	3.03 us.	1.58 us.
7.07 us.	2.96 us.	1.58 us.
7.64 us.	3.03 us.	1.65 us.
7.20 us.	3.10 us.	1.58 us.
7.26 us.	7.96 us.	1.65 us.
7.25 us.	6.06 us.	1.65 us.
7.19 us.	3.28 us.	2.08 us.
7.20 us.	3.28 us.	1.65 us.
7.07 us.	3.28 us.	1.58 us.
7.07 us.	3.09 us.	1.59 us.
7.77 us.	3.22 us.	1.52 us.
7.07 us.	3.22 us.	1.58 us.
7.38 us.	3.15 us.	1.58 us.
7.07 us.	3.47 us.	1.59 us.
7.38 us.	3.22 us.	1.59 us.
8.27 us.	4.29 us.	1.58 us.
7.44 us.	3.34 us.	1.58 us.
7.38 us.	3.34 us.	1.58 us.
8.08 us.	3.09 us.	1.58 us.
7.07 us.	3.15 us.	1.58 us.
7.01 us.	3.28 us.	1.58 us.
7.26 us.	3.09 us.	1.65 us.
7.14 us.	3.09 us.	1.64 us.
7.13 us.	3.15 us.	1.58 us.
7.13 us.	3.09 us.	1.64 us.
7.01 us.	3.10 us.	1.59 us.
MEAN: 7,32566 us.	MEAN: 3,483666 us.	MEAN: 1,709333 us.
MAX: 8,08 us.	MAX: 7,96 us.	MAX: 2,66 us.
MIN: 7,01 us.	MIN: 2,96 us.	MIN: 1,58 us.

In this values I computed mean (only on the last 30 elements, I discarded the first 2 elements because in the relu_plain they were too high, I think that the first 1 or 2 times are higher because of cache fill-in time), max value and min value. I have noticed another interesting thing: the cycle 11-12 takes much more times than the other only on relu_NEON execution but I cannot explain why.

As I could imagine the ReLU plain autovectorized took much less time than the other, this means that the optimization made by the compiler is much more efficient than the one made by me.