Group 6: Video Encoding on ARMv8 using ARM NEON Instructions

Skjalg Gustav Eriksen
University of Oslo
skjale@ifi.uio.no

ABSTRACT

In this paper we document our optimization of the Codec63 encoder. We will optimize the encoder with the use of the neon parallelization options in ARMv8 to accelerate Codec63.

1 INTRODUCTION

This paper is the home exam 1 for group 6 in IN5050 at UiO. The assignment is to take advantage of the parallelization options available in a single ARMv8 NEON-enabled core to accelerate Codec63. During the optimization process we will keep the -p cflag on so we can see the changes to the gprof profile and document bottlenecks that may arise.

1.1 Encoder profile

Figure 1 shows the flat profile from gprof when compiling and ruining the unmodified code. It clearly shows that sad_block_8x8 should be the priority when optimizing as the program spends over 90% of its time in that function. after sad_block_8x8 is optimized the dct functions; dct_1d and idct_1d are the next heaviest load and should be looked at.

1.2 Sections

The remainder of this paper is organized in five sections. Section 2 we document the optimizations for the function sad_block_8x8 and discuss the results. Section 3 we discuss the optimization done on dct_1d and idct_1d. Section 4 we summarize the results of our optimizations. Section 5 concludes the paper.

2 SAD_BLOCK_8X8

The function sad_block_8x8 is used to calculate the Sum Absolute Difference(SAD) in two 8 by 8 blocks from the video frames and is used to create the motion vectors in the encoded video.

Since the function uses a nested for-loop with a inner and outer loop running 8 times each, its easy to attack and vectorize with Single Instruction Multiple Data (SIMD) neon intrinsics. With SIMD you can do one instruction on multiple data which can be used to calculates the SAD of 8 elements from each block with a single instruction rather than 8.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 92.87     43.12    43.12 500214528     0.00     0.00  sad_block_8x8
  1.57     43.85     0.73 11404800     0.00     0.00  dct_1d
  1.42     44.51     0.66 11404800     0.00     0.00  idct_1d
  1.08     45.01     0.50   705672     0.00     0.06  me_block_8x8
  0.54     45.26     0.25  2851200     0.00     0.00  transpose_block
  0.45     45.47     0.21 16398942     0.00     0.00  put_bits
  0.45     45.68     0.21   712800     0.00     0.00  dct_quant_block_8x8
  0.43     45.88     0.20  1425600     0.00     0.00  scale_block
  0.28     46.01     0.13   712800     0.00     0.00  dequant_idct_block_8x8
  0.26     46.13     0.12   356400     0.00     0.00  write_interleaved_data_MCU
  0.24     46.24     0.11  4752503     0.00     0.00  put_byte
  0.17     46.32     0.08   705672     0.00     0.00  mc_block_8x8
  0.17     46.40     0.08    21600     0.00     0.06  dct_quantize_row
  0.06     46.43     0.03    21600     0.00     0.05  dequantize_idct_row
  0.00     46.43     0.00     3300     0.00     0.00  put_bytes
  0.00     46.43     0.00     1200     0.00     0.00  write_DHT_HTS
  0.00     46.43     0.00      900     0.00     1.38  dct_quantize
  0.00     46.43     0.00      900     0.00     1.16  dequantize_idct
  0.00     46.43     0.00      301     0.00     0.00  destroy_frame
  0.00     46.43     0.00      300     0.00     0.00  create_frame
  0.00     46.43     0.00      300     0.00     0.00  flush_bits
  0.00     46.43     0.00      300     0.00     1.47  write_frame
  0.00     46.43     0.00      297     0.00     0.27  c63_motion_compensate
  0.00     46.43     0.00      297     0.00   146.87  c63_motion_estimate
  0.00     46.43     0.00        1     0.00     0.00  free_c63_enc
  0.00     46.43     0.00        1     0.00     0.00  init_c63_enc
```

Figure 1: initial profile with gprof

When optimizing sad_block_8x8 we started out trying to load 8 elements from a block with neon intrinsics, to find the appropriate instruction we looked up vld instructions in the neon reference documentation [1]. The input types for the Blocks was uint8_t and we quickly realized the pattern uint8_t related instructions had _u8 at the end to indicate unsigned 8-bit integer. We found the instruction `vld1_u8()` to load 8 elements from our uint8_t block into a uint8x8_t vector. Using the same procedure as before we found the instruction `vst1_u8()` to store the variable back into a uint8_t. While looking at the documentation we also discovered the convenient instruction `vaddv_u()` which calculates a vector wide sum that could be added directly to the SAD result int. Now we only needed a way to calculate the difference of two loaded vectors.

Since the type is unsinged and does not have negative values we thought at first just to use `vsub_u8()` and subtract the two vectors then add them to the SAD result int with `vaddv_u()`. We then found something had gone wrong when comparing the SAD outputs with the unmodified code. Thinking it was the `vsub_u8()` having unexpected results as it did subtraction instead of absolute difference, we tried to use `vabd_u8()` (Unsigned absolute difference) instead. When this also failed we looked closely at the outputs and found that summing up the 8 elements could easily exceed the maximum size of 255 of our 8-bit uints.

Looking for remedies in the documentation we found `vabdl_u8()` (Unsigned absolute difference long), this instruction outputs a half-word (16-bit) instead of a uint8_t (8-bit). After adjusting the vector sum instruction to fit 16-bit with `vaddvq_u16()` the code ran without issue.

After this initial optimization of sad_block_8x8 we timed our code and went from `0m41,499s` to `0m26,053s` real time.

The this optimized version of sad_block_8x8 has does the following in-place of the inner loop; load 8 elements from block1 and

```
void sad_block_8x8(uint8_t *block1, uint8_t *block2, int
    stride, int *result)
{
// variables hold 8 block1 and block2 elements
uint8x8_t b_1, b_2;
// variables calcualte sad and hold total sad.
uint16x8_t sad, total_sad;
*result = 0;

for (v = 0; v < 8; ++v)
    {
        /* Neon Intrinsics */
        // load 8 elems from block1 and block2
        b_1 = vld1_u8(block1 + v*stride);
        b_2 = vld1_u8(block2 + v*stride);

        // calculate abs difference long, uint8x8_t ->
    uint16x8_t
        sad = vabdl_u8(b_2, b_1);
        // vector wide sum
        *result += vaddvq_u16(sad);
    }
}
```

Figure 2: sad_block_8x8 initial neon optimization

```
void sad_block_8x8(uint8_t *block1, uint8_t *block2, int
    stride, int *result)
{
// variables hold 8 block1 and block2 elements
uint8x8_t b_1, b_2;
// variables calcualte sad and hold total sad.
uint16x8_t sad, total_sad;
*result = 0;

for (v = 0; v < 8; ++v)
    {
        /* Neon Intrinsics */
        // load 8 elems from block1 and block2
        b_1 = vld1_u8(block1 + v*stride);
        b_2 = vld1_u8(block2 + v*stride);

        // calculate abs difference long, uint8x8_t ->
    uint16x8_t
        sad = vabdl_u8(b_2, b_1);
        // add to total sum
        total_sad = vaddq_u16(sad, total_sad);
    }
    // vector wide sum
    *result += vaddvq_u16(total_sad);
}
```

Figure 3: sad_block_8x8 neon optimization improved

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 77.76      9.86     9.86 500214528     0.00     0.00  sad_block_8x8
  5.99     10.62     0.76 11404800     0.00     0.00  dct_1d
  4.57     11.20     0.58 11404800     0.00     0.00  idct_1d
  2.60     11.53     0.33  2851200     0.00     0.00  transpose_block
  2.29     11.82     0.29 16398942     0.00     0.00  put_bits
  2.05     12.08     0.26   712800     0.00     0.00  dct_quant_block_8x8
  1.74     12.30     0.22  1425600     0.00     0.00  scale_block
  0.87     12.41     0.11  4752503     0.00     0.00  put_byte
  0.63     12.49     0.08   705672     0.00     0.01  me_block_8x8
  0.47     12.55     0.06   705672     0.00     0.00  mc_block_8x8
  0.39     12.60     0.05   712800     0.00     0.00  dequant_idct_block_8x8
  0.39     12.65     0.05    21600     0.00     0.04  dequantize_idct_row
  0.24     12.68     0.03   356400     0.00     0.00  write_interleaved_data_MCU
  0.00     12.68     0.00    21600     0.00     0.06  dct_quantize_row
  0.00     12.68     0.00     3300     0.00     0.00  put_bytes
  0.00     12.68     0.00     1200     0.00     0.00  write_DHT_HTS
  0.00     12.68     0.00      900     0.00     1.44  dct_quantize
  0.00     12.68     0.00      900     0.00     1.06  dequantize_idct
  0.00     12.68     0.00      301     0.00     0.00  destroy_frame
  0.00     12.68     0.00      300     0.00     0.00  create_frame
  0.00     12.68     0.00      300     0.00     0.00  flush_bits
  0.00     12.68     0.00      300     0.00     1.43  write_frame
  0.00     12.68     0.00      297     0.00     0.20  c63_motion_compensate
  0.00     12.68     0.00      297     0.00    33.47  c63_motion_estimate
  0.00     12.68     0.00        1     0.00     0.00  free_c63_enc
  0.00     12.68     0.00        1     0.00     0.00  init_c63_enc

 %         the percentage of the total running time of the
time       program used by this function.
```

Figure 4: profile after optimizing sad_block_8x8

block2, then calculates the absolute difference, then does the vector wide sum and adds it to the SAD result int as shown in figure 2. This code moves the values out of the neon variables for every iteration of the outer loop and we suspected this to be slow. So to change this we added one more neon variable to sum the absolute difference with `vaddq_u16` (addition) then add the total sad value to the non-neon SAD result int after exiting the outer loop as show in figure 3. This gave further improvements `0m26,053s` to `0m20,078s` real time.

The last thing we thought of doing was unrolling the outer loop, in theory this will make it go a bit faster due to not having to do a variable check each iteration. This final improvement gave the time improvement `0m20,078s` to `0m18,972s` real time.

While sad_block_8x8 is still at the top of the gprof profile shown in figure 4 but compared to figure 1 with the initial profile of the unmodified code, the self seconds is about 4 times smaller after neon optimization.

2.1 inline assembly

After the initial implementation of neon intrinsics shown in figure 2 we looked up the assembly equivalent instructions in the documentation [1] and tried to implement our solution in inline assembly. After we figured out the assembly structure it was not complicated to convert our neon intrinsics into working inline assembly code shown in figure 5. This code replaces the neon intrinsics in figure 2. Performance boost in speed going from the initial neon intrinsics to inline-assembly gave `0m26,053s` to `0m24,357s`.

We decided against using inline assembly because the intrinsics were easier to work with.

3 DCT_1D AND IDCT_1D

The dct_1d and idct_1d functions calculate a Discrete Cosine Transform (dct), in Codec63 the dct functions are implemented with the use of a pre-calculated table (dctlookup) we can lookup to find the right transformation to apply to our input data (in_data). the difference in dct_1d and idct_1d just that dct_1d effectively uses the dctlookup table or the in_data transposed. This means we can use the same optimization for both functions by just transposing the in_data in dct_1d.

Much like sad_block_8x8 the dct functions are implemented with nested for-loops that has a inner and outer loop run 8 times each. With what we learnt optimizing sad_block_8x8 in section 2 we started by looking for a way to load the input variables to neon variables.

The input variables for the dct functions are float and unfortunately they are too big to load 8 float elements into one neon variable so we had to settle loading 4 elements at a time in a float32x4_t. There was also a instruction for loading float32x4x2_t which would cover the 8 elements but we could not get this instruction to work. So we ended up loading 2 separate float32x4_t variables to hold our 8 elements. To do that we used the neon instruction `vld1q_f32()`.

```
uint8_t *blk_1;
uint8_t *blk_2;

blk_1 = block1 + v*stride;
blk_2 = block2 + v*stride;

__asm__ (
  "ld1 {v0.8h}, [%0]\n\t" // load result value
  "ld1 {v1.8b}, [%1]\n\t" // load block1
  "ld1 {v2.8b}, [%2]\n\t" // load block2

   // calculate absolute difference long
  "uabdl v3.8h, v1.8b, v2.8b\n\t"
  // vector wide sum store in register v3
  "addv h3, v3.8h\n\t"
  // add v3 to result value
  "add v0.8h, v0.8h, v3.8h\n\t"
   // store result value
  "st1 {v0.8h}, [%0]\n\t"

: // output
: "r" (result), "r" (blk_1), "r" (blk_2) // input
: "v0", "v1", "v2", "v3", "memory"  // dirty registers
);
```

Figure 5: sad_block_8x8 inline assembly

The in_data would be loaded before the nested loop. Then inside the outer loop we will replace the inner loop as we did in sad_block_8x8 by loading in 8 elements (2x float32x4_t) from the dctlookup. From the reference documentation [1] we find that `vmulq_f32()` is the appropriate neon instruction to multiply two float32x4_t variables and we use that to multiply the in_data with the dctlookup neon variables. Once we have multiplied both in_data variables (2x float32x4_t) with both dctlookup variables (2x float32x4_t) we add them together with `vaddq_f32()`(addition) and sum together that vector with the vector wide addition `vaddvq_f32()` and store that in the out_data. This is shown in figure 6.

dct_1d uses the same implementation but with a transposed input data using the function transpose_block before its loaded into neon variables.

This initial optimization attempt reduced the speed of the code and gave `0m18,972s` to `0m21,384s` real time. looking at gprof's flat profile in figure 7 this is explained by the function transpose_block being used for each call of dct_1d and slowing the code down by being a new bottle neck.

The quick solution this is simply by making a transposed version of the dctlookup table in tables.c alongside the normal dctlookup that we can use in dct_1d that will save us from using transpose_block for each call of dct_1d.

After making a pre-transposed dctlookup table for dct_1d we get the expected 'small' improvement from the neon optimization from `0m18,972s` to `0m18,394s` real time.

Final optimization for the dct functions is to unroll their loops like we did in sad_block_8x8. unrolling the dct loops gave another little small speed up from `0m18,394s` to `0m17,870s` real time.

In Figure 8 we see improvements in dct_1d and idct_1d when we compare their self seconds to our initial profile on the unmodified code in figure 1 and see that the time spent in them is roughly halved.

```
  // load first 4 elements from in_data
  in = vld1q_f32 (in_data);
  // load last 4 elements from in_data
  in2 = vld1q_f32 (in_data +4);

  for (i = 0; i < 8; ++i)
  {
    // load lookup table into neon varriables
    float32x4_t lookup = vld1q_f32 ((dctlookup+i));
    float32x4_t lookup2 = vld1q_f32 ((dctlookup[i])+4);

    // multiply first 4 elements with with first 4
     elements of the lookuptable
    r = vmulq_f32(in, lookup);
    // multiply last 4 elements with with first 4
     elements of the lookuptable
    r2 = vmulq_f32(in2, lookup2);
    // add up the 4 elements from r and r2
    r = vaddq_f32(r, r2);

    // add vector wide sum to out_data
    out_data[i] = vaddvq_f32(r);
    }
```

Figure 6: idct 1d initial optimization

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 73.94     11.32    11.32 500214528     0.00     0.00  sad_block_8x8
  9.21     12.73     1.41 14256000     0.00     0.00  transpose_block
  5.55     13.58     0.85 66182646     0.00     0.00  put_bits
  3.53     14.12     0.54 25527122     0.00     0.00  put_byte
  2.42     14.49     0.37 11404800     0.00     0.00  idct_1d
  2.02     14.80     0.31   712800     0.00     0.00  dct_quant_block_8x8
  1.18     14.98     0.18   712800     0.00     0.00  dequant_idct_block_8x8
  0.52     15.06     0.08 11404800     0.00     0.00  dct_1d
  0.39     15.12     0.06   705672     0.00     0.00  mc_block_8x8
  0.39     15.18     0.06   356400     0.00     0.00  write_interleaved_data_MCU
  0.33     15.23     0.05  1425600     0.00     0.00  scale_block
  0.33     15.28     0.05    21600     0.00     0.04  dequantize_idct_row
  0.13     15.30     0.02    21600     0.00     0.08  dct_quantize_row
  0.07     15.31     0.01      297     0.03     0.24  c63_motion_compensate
  0.00     15.31     0.00   705672     0.00     0.02  me_block_8x8
  0.00     15.31     0.00     3300     0.00     0.00  put_bytes
  0.00     15.31     0.00     1200     0.00     0.00  write_DHT_HTS
  0.00     15.31     0.00      900     0.00     1.89  dct_quantize
  0.00     15.31     0.00      900     0.00     0.85  dequantize_idct
  0.00     15.31     0.00      301     0.00     0.00  destroy_frame
  0.00     15.31     0.00      300     0.00     0.00  create_frame
  0.00     15.31     0.00      300     0.00     0.00  flush_bits
  0.00     15.31     0.00      300     0.00     4.83  write_frame
  0.00     15.31     0.00      297     0.00    38.11  c63_motion_estimate
  0.00     15.31     0.00        1     0.00     0.00  free_c63_enc
  0.00     15.31     0.00        1     0.00     0.00  init_c63_enc
```

Figure 7: profile after inital attempt at optimizing dct

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 84.35     11.60    11.60 500214528     0.00     0.00  sad_block_8x8
  2.47     11.94     0.34 11404800     0.00     0.00  idct_1d
  2.47     12.28     0.34  2851200     0.00     0.00  transpose_block
  2.15     12.57     0.30 11404800     0.00     0.00  dct_1d
  1.75     12.81     0.24   712800     0.00     0.00  dct_quant_block_8x8
  1.67     13.04     0.23   712800     0.00     0.00  dequant_idct_block_8x8
  1.16     13.20     0.16 16397176     0.00     0.00  put_bits
  1.13     13.36     0.16  1425600     0.00     0.00  scale_block
  0.73     13.46     0.10  4753139     0.00     0.00  put_byte
  0.51     13.53     0.07   356400     0.00     0.00  write_interleaved_data_MCU
  0.51     13.60     0.07    21600     0.00     0.04  dequantize_idct_row
  0.51     13.67     0.07                             read_bytes
  0.44     13.73     0.06   705672     0.00     0.00  mc_block_8x8
  0.07     13.74     0.01   705672     0.00     0.02  me_block_8x8
  0.07     13.75     0.01      300     0.03     1.13  write_frame
  0.07     13.76     0.01      297     0.03    39.12  c63_motion_estimate
  0.00     13.76     0.00    21600     0.00     0.04  dct_quantize_row
  0.00     13.76     0.00     3300     0.00     0.00  put_bytes
  0.00     13.76     0.00     1200     0.00     0.00  write_DHT_HTS
  0.00     13.76     0.00      900     0.00     0.87  dct_quantize
  0.00     13.76     0.00      900     0.00     0.99  dequantize_idct
  0.00     13.76     0.00      301     0.00     0.00  destroy_frame
  0.00     13.76     0.00      300     0.00     0.00  create_frame
  0.00     13.76     0.00      300     0.00     0.00  flush_bits
  0.00     13.76     0.00      297     0.00     0.20  c63_motion_compensate
  0.00     13.76     0.00        1     0.00     0.00  free_c63_enc
  0.00     13.76     0.00        1     0.00     0.00  init_c63_enc
```

Figure 8: profile after optimizing dct

4 EVALUATION

In sad_block_8x8 we neonized the inner loop allowing the computer to calculate the absolute difference between two 8 element vectors in parallel. We then unrolled the outer loop to remove unnecessary variable checks. this gave us a performance increase from `0m41,499s` to `0m18,972s` real time, this is about 54% faster.

For the dct functions dct_1d and idct_1d we neonized their inner loops allowing the computer to multiply two 4 element vectors in parallel. And as before unrolled the outer loop. this gave us from `0m18,972s` to `0m17,870s`, giving us another `5,8%` speed increase.

After removing the -p from the cflags we get the final result `0m17,870s` to `0m6,693s` real time.

in total from `0m41,499s` to `0m6,693s`, our optimizations has shaved off `34,806s` of run-time which makes it about `83,9%` faster.

5 CONCLUSION

In This assignment we have learnt that replacing loops with SIMD neon variables to run a instruction on multiple data in parallel give a great performance boost. I've also been surprised at the performance boost from unrolling loops, i had not expected to see any visible improvements.

REFERENCES

[1] arm. Neon Intrinsics Reference. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics/. (????). [Online; accessed 01-march-2021].