ECE785 – Project 3A: Code Optimization

Name: Zhiyu Chen

UnityID: zchen45

1. Overview

In this project, I am supposed to profile the execution time for the battery modeling program from Dr. Chow’s research group and then optimize it within a limited amount of time. Obtain and analyze the execution time profile of the CoEst_offline.c program.

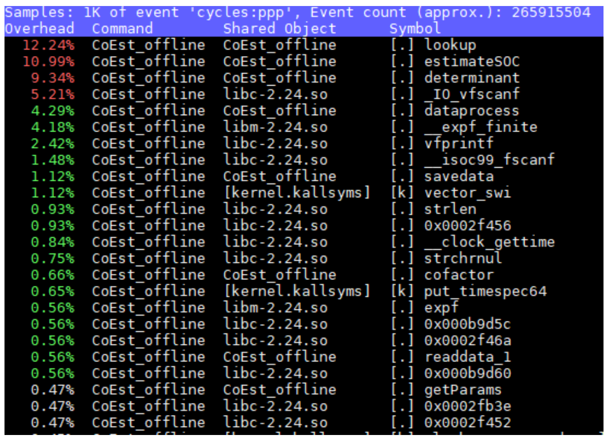
2. Optimization

2.1. Initial profile

In this project, I use the clock_gettime to measure the time duration. And the duration to be measured is between “fopen” and “fclose”. By the command “gcc CoEst_offline.c –o CoEst_offline –lm; ./CoEst_offline”, the results are listed below.

Time: 210890.500 us

The Figure 1 is the perf report. As we can see, about 33 percent of overheads occurs in the function lookup, estimateSOC and determinant.



```
Samples: 1K of event 'cycles:ppp', Event count (approx.): 265915504
Overhead  Command        Shared Object       Symbol
  12.24%  CoEst_offline  CoEst_offline       [.] lookup
  10.99%  CoEst_offline  CoEst_offline       [.] estimateSOC
   9.34%  CoEst_offline  CoEst_offline       [.] determinant
   5.21%  CoEst_offline  libc-2.24.so        [.] _IO_vfscanf
   4.29%  CoEst_offline  CoEst_offline       [.] dataprocess
   4.18%  CoEst_offline  libm-2.24.so        [.] __expf_finite
   2.42%  CoEst_offline  libc-2.24.so        [.] vfprintf
   1.48%  CoEst_offline  libc-2.24.so        [.] __isoc99_fscanf
   1.12%  CoEst_offline  CoEst_offline       [.] savedata
   1.12%  CoEst_offline  [kernel.kallsyms]   [k] vector_swi
   0.93%  CoEst_offline  libc-2.24.so        [.] strlen
   0.93%  CoEst_offline  libc-2.24.so        [.] 0x0002f456
   0.84%  CoEst_offline  libc-2.24.so        [.] __clock_gettime
   0.75%  CoEst_offline  libc-2.24.so        [.] strchrnul
   0.66%  CoEst_offline  CoEst_offline       [.] cofactor
   0.65%  CoEst_offline  [kernel.kallsyms]   [k] put_timespec64
   0.56%  CoEst_offline  libm-2.24.so        [.] expf
   0.56%  CoEst_offline  libc-2.24.so        [.] 0x000b9d5c
   0.56%  CoEst_offline  libc-2.24.so        [.] 0x0002f46a
   0.56%  CoEst_offline  CoEst_offline       [.] readdata_1
   0.56%  CoEst_offline  libc-2.24.so        [.] 0x000b9d60
   0.47%  CoEst_offline  CoEst_offline       [.] getParams
   0.47%  CoEst_offline  libc-2.24.so        [.] 0x0002fb3e
   0.47%  CoEst_offline  libc-2.24.so        [.] 0x0002f452
```

Figure 1   Perf Report 1

2.2. Compiler flags

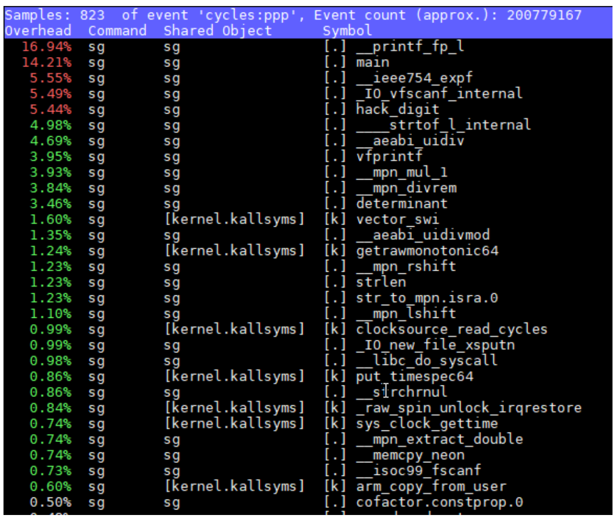
The first step is to set the compiler flags. This is a simple but effective way to improve the performance. Also it is necessary for vectorizing the code in the future. The follows show the flags implemented.

Flags:

-mfpu=neon -mfloat-abi=hard -mcpu=cortex-a8 -mtune=cortex-a8 -ftree-vectorize -mvectorize-with-neon-quad -O3 -g -pg -ffast-math

Time: 149865.708 us

Perf report:

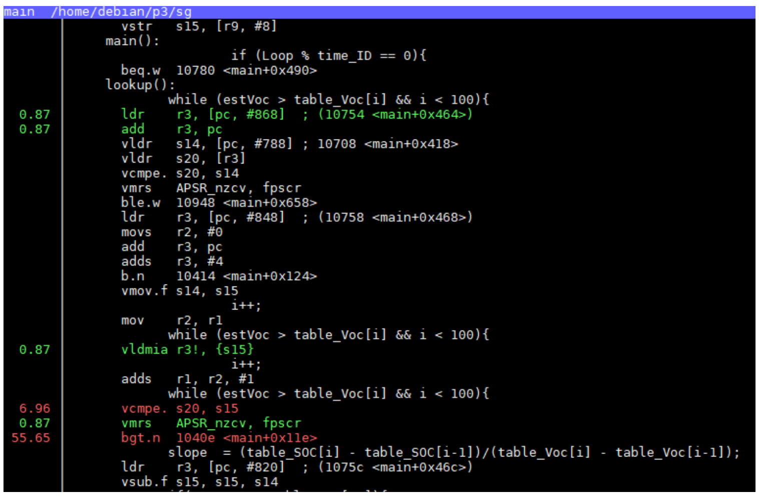


```
Samples: 823  of event 'cycles:ppp', Event count (approx.): 200779167
Overhead  Command  Shared Object      Symbol
  16.94%  sg       sg                 [.] __printf_fp_l
  14.21%  sg       sg                 [.] main
   5.55%  sg       sg                 [.] __ieee754_expf
   5.49%  sg       sg                 [.] _IO_vfscanf_internal
   5.44%  sg       sg                 [.] hack_digit
   4.98%  sg       sg                 [.] ____strtof_l_internal
   4.69%  sg       sg                 [.] __aeabi_uidiv
   3.95%  sg       sg                 [.] vfprintf
   3.93%  sg       sg                 [.] __mpn_mul_1
   3.84%  sg       sg                 [.] __mpn_divrem
   3.46%  sg       sg                 [.] determinant
   1.60%  sg       [kernel.kallsyms]  [k] vector_swi
   1.35%  sg       sg                 [.] __aeabi_uidivmod
   1.24%  sg       [kernel.kallsyms]  [k] getrawmonotonic64
   1.23%  sg       sg                 [.] __mpn_rshift
   1.23%  sg       sg                 [.] strlen
   1.23%  sg       sg                 [.] str_to_mpn.isra.0
   1.10%  sg       sg                 [.] __mpn_lshift
   0.99%  sg       [kernel.kallsyms]  [k] clocksource_read_cycles
   0.99%  sg       sg                 [.] _IO_new_file_xsputn
   0.98%  sg       sg                 [.] __libc_do_syscall
   0.86%  sg       [kernel.kallsyms]  [k] put_timespec64
   0.86%  sg       sg                 [.] __strchrnul
   0.84%  sg       [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
   0.74%  sg       [kernel.kallsyms]  [k] sys_clock_gettime
   0.74%  sg       sg                 [.] __mpn_extract_double
   0.74%  sg       sg                 [.] __memcpy_neon
   0.73%  sg       sg                 [.] __isoc99_fscanf
   0.60%  sg       [kernel.kallsyms]  [k] arm_copy_from_user
   0.50%  sg       sg                 [.] cofactor.constprop.0
```

Figure 2   Perf Report 2

2.3. Lookup function

As we can see in Figure 1, lookup function generates the most overheads. Also by annotating the main function in the perf report 2, the lookup function produces more than 50 percent of overheads as Figure 3 shown below. So the next step is to optimize the lookup function.



```
main  /home/debian/p3/sg
       │        vstr   s15, [r9, #8]
       │      main():
       │                       if (Loop % time_ID == 0){
       │        beq.w  10780 <main+0x490>
       │      lookup():
       │                while (estVoc > table_Voc[i] && i < 100){
  0.87 │        ldr    r3, [pc, #868]  ; (10754 <main+0x464>)
  0.87 │        add    r3, pc
       │        vldr   s14, [pc, #788] ; 10708 <main+0x418>
       │        vldr   s20, [r3]
       │        vcmpe. s20, s14
       │        vmrs   APSR_nzcv, fpscr
       │        ble.w  10948 <main+0x658>
       │        ldr    r3, [pc, #848]  ; (10758 <main+0x468>)
       │        movs   r2, #0
       │        add    r3, pc
       │        adds   r3, #4
       │        b.n    10414 <main+0x124>
       │        vmov.f s14, s15
       │                       i++;
       │        mov    r2, r1
       │                while (estVoc > table_Voc[i] && i < 100){
  0.87 │        vldmia r3!, {s15}
       │                       i++;
       │        adds   r1, r2, #1
       │                while (estVoc > table_Voc[i] && i < 100){
  6.96 │        vcmpe. s20, s15
  0.87 │        vmrs   APSR_nzcv, fpscr
 55.65 │        bgt.n  1040e <main+0x11e>
       │                slope  = (table_SOC[i] - table_SOC[i-1])/(table_Voc[i] - table_Voc[i-1]);
       │        ldr    r3, [pc, #820]  ; (1075c <main+0x46c>)
       │        vsub.f s15, s15, s14
```

Figure 3   Annotate main in perf report 2

In the annotation, it is obvious that the while loop generates lots of overhead.

```c
int i = 0; // index of look-up table
float slope = 0.0;
// find the index
while (estVoc > table_Voc[i] && i < 100){
    i++;
}
```

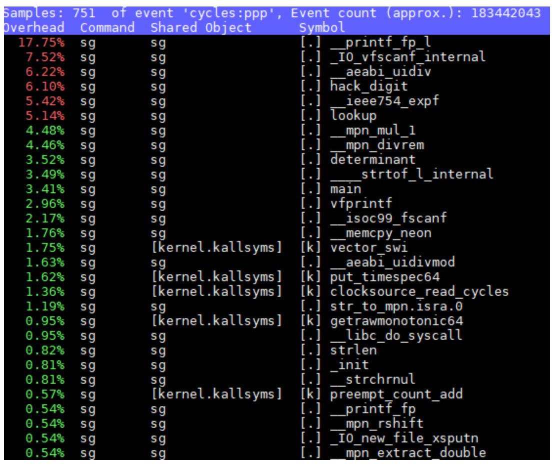
Every time when lookup function is called, it will compare estVoc with the table of voc, from low index to high index, until the estVoc is bigger than the voc or it compares 100 times. According to the figure 3, there is a red line leading to 55 percent of overheads. It is the conditional branch of the while loop. The optimized code is shown below.

```c
float32x4_t n_index, n_index_2, n_index_3, t_voc;
uint32x4_t mask_i = vmovq_n_u32(0);
uint32x4_t change_or_not;
float32x4_t index_add = {1.0f, 2.0f, 3.0f, 4.0f};
float32x4_t estVoc_n = vld1q_dup_f32(&estVoc);

for(int a = 0; a < 100; a = a + 4){
  index = (float) a;
  n_index = vdupq_n_f32(index);
  n_index = vaddq_f32(n_index,index_add);

  t_voc = vld1q_f32(&table_Voc[a]);
  mask_i = vcgtq_f32(estVoc_n, t_voc);
  n_index_3 = n_index_2;
  n_index_2 = vbslq_f32(mask_i, n_index, n_index_2);
  n_index_2 = vmaxq_f32(n_index_2, n_index_3);
}
```

As the conditional branch produces too much overheads, I decide to vectorize the code to unroll the while loop. At here, I compare the estVoc with all the elements in voc table and record the element with the smallest index.

Time: 134308.416 us

Perf report:



```
Samples: 751  of event 'cycles:ppp', Event count (approx.): 183442043
Overhead  Command  Shared Object      Symbol
  17.75%  sg       sg                 [.] __printf_fp_l
   7.52%  sg       sg                 [.] _IO_vfscanf_internal
   6.22%  sg       sg                 [.] __aeabi_uidiv
   6.10%  sg       sg                 [.] hack_digit
   5.42%  sg       sg                 [.] __ieee754_expf
   5.14%  sg       sg                 [.] lookup
   4.48%  sg       sg                 [.] __mpn_mul_1
   4.46%  sg       sg                 [.] __mpn_divrem
   3.52%  sg       sg                 [.] determinant
   3.49%  sg       sg                 [.] ____strtof_l_internal
   3.41%  sg       sg                 [.] main
   2.96%  sg       sg                 [.] vfprintf
   2.17%  sg       sg                 [.] __isoc99_fscanf
   1.76%  sg       sg                 [.] __memcpy_neon
   1.75%  sg       [kernel.kallsyms]  [k] vector_swi
   1.63%  sg       sg                 [.] __aeabi_uidivmod
   1.62%  sg       [kernel.kallsyms]  [k] put_timespec64
   1.36%  sg       [kernel.kallsyms]  [k] clocksource_read_cycles
   1.19%  sg       sg                 [.] str_to_mpn.isra.0
   0.95%  sg       [kernel.kallsyms]  [k] getrawmonotonic64
   0.95%  sg       sg                 [.] __libc_do_syscall
   0.82%  sg       sg                 [.] strlen
   0.81%  sg       sg                 [.] _init
   0.81%  sg       sg                 [.] __strchrnul
   0.57%  sg       [kernel.kallsyms]  [k] preempt_count_add
   0.54%  sg       sg                 [.] __printf_fp
   0.54%  sg       sg                 [.] __mpn_rshift
   0.54%  sg       sg                 [.] _IO_new_file_xsputn
   0.54%  sg       sg                 [.] __mpn_extract_double
```

Figure 4   perf report 3

2.4. Matrix calculation optimization

In the original code, it implements a lot of matrix multiplication. The follow is an example.

```c
// Step 7: Calculate Ad^2 =  A * A
for (int i = 0; i < tforder; i++){
    for (int j = 0; j < tforder; j++){
        AA[i][j] = 0;
        for (int k = 0; k < tforder; k++)
            AA[i][j] += Ad[i][k] * Ad[k][j];
    }
}
```

It implements 3 for-loops to achieve the multiplication of 2 2x2 matrixes. Actually how we calculate it on paper?

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a00b00 + a01b10 & a00b01 + a01b11 \\ a10b00 + a11b10 & a10b01 + a11b11 \end{bmatrix}$$

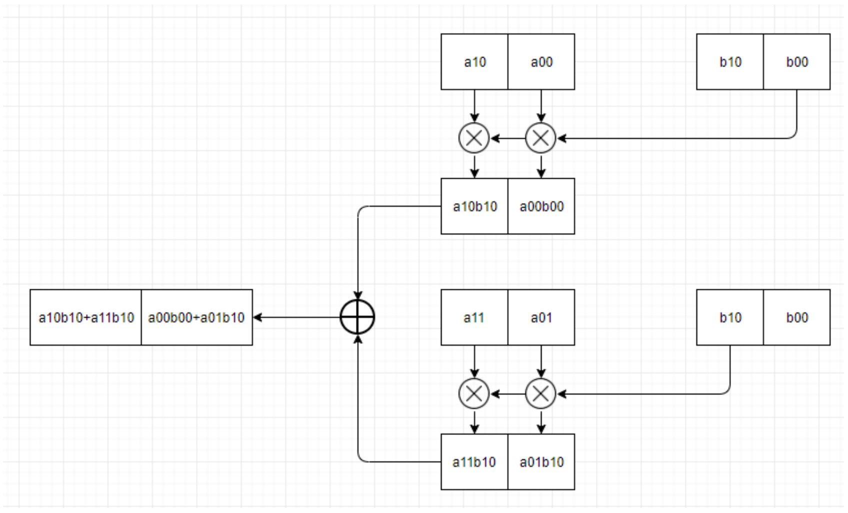


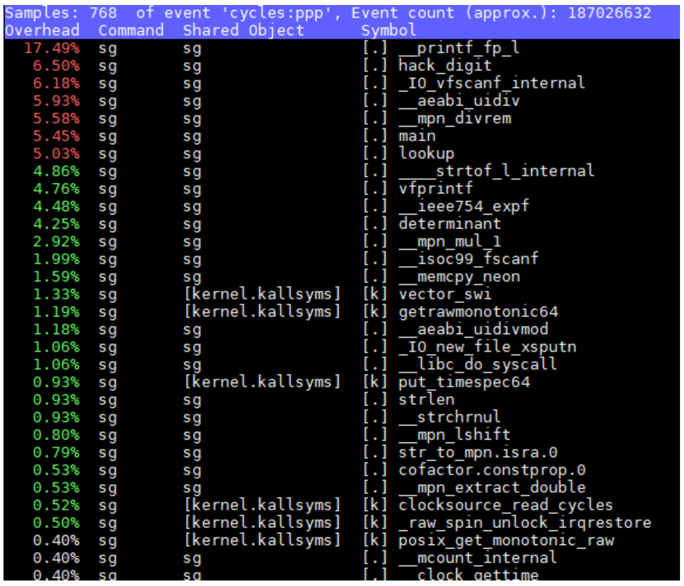
Figure 5 Matrix Multiplication

If each column of matrixes is a vector in NEON register, we can use the vector-by-scalar-multiplication to calculate each result column efficiently. The optimized code is shown below.

```c
// Step 7: Calculate Ad^2 =  A * A
float temp[4] = {Ad[0][0], Ad[1][0], Ad[0][1], Ad[1][1]};
float32x2_t Ad_n0 = vld1_f32(&temp[0]);
float32x2_t Ad_n1 = vld1_f32(&temp[2]);

float32x2_t AA_n0 = vadd_f32(vmul_n_f32(Ad_n0, Ad_n0[0]), vmul_n_f32(Ad_n1, Ad_n0[1]));
float32x2_t AA_n1 = vadd_f32(vmul_n_f32(Ad_n0, Ad_n1[0]), vmul_n_f32(Ad_n1, Ad_n1[1]));
```

Time: 136094.291 us

Perf report:



```
Samples: 768  of event 'cycles:ppp', Event count (approx.): 187026632
Overhead  Command  Shared Object       Symbol
  17.49%  sg       sg                  [.] __printf_fp_l
   6.50%  sg       sg                  [.] hack_digit
   6.18%  sg       sg                  [.] _IO_vfscanf_internal
   5.93%  sg       sg                  [.] __aeabi_uidiv
   5.58%  sg       sg                  [.] __mpn_divrem
   5.45%  sg       sg                  [.] main
   5.03%  sg       sg                  [.] lookup
   4.86%  sg       sg                  [.] ____strtof_l_internal
   4.76%  sg       sg                  [.] vfprintf
   4.48%  sg       sg                  [.] __ieee754_expf
   4.25%  sg       sg                  [.] determinant
   2.92%  sg       sg                  [.] __mpn_mul_1
   1.99%  sg       sg                  [.] __isoc99_fscanf
   1.59%  sg       sg                  [.] __memcpy_neon
   1.33%  sg       [kernel.kallsyms]   [k] vector_swi
   1.19%  sg       [kernel.kallsyms]   [k] getrawmonotonic64
   1.18%  sg       sg                  [.] __aeabi_uidivmod
   1.06%  sg       sg                  [.] _IO_new_file_xsputn
   1.06%  sg       sg                  [.] __libc_do_syscall
   0.93%  sg       [kernel.kallsyms]   [k] put_timespec64
   0.93%  sg       sg                  [.] strlen
   0.93%  sg       sg                  [.] __strchrnul
   0.80%  sg       sg                  [.] __mpn_lshift
   0.79%  sg       sg                  [.] str_to_mpn.isra.0
   0.53%  sg       sg                  [.] cofactor.constprop.0
   0.53%  sg       sg                  [.] __mpn_extract_double
   0.52%  sg       [kernel.kallsyms]   [k] clocksource_read_cycles
   0.50%  sg       [kernel.kallsyms]   [k] _raw_spin_unlock_irqrestore
   0.40%  sg       [kernel.kallsyms]   [k] posix_get_monotonic_raw
   0.40%  sg       sg                  [.] __mcount_internal
   0.40%  sg       sg                  [.] __clock_gettime
```

Figure 6   Perf Report 4

But the result doesn’t meet my expectation. The time of duration increases. Maybe it is influenced by the size of the for-loop. As it is only a 2x2 matrix, the benefit from vectorizing is small. Also, it is said that the NEON is in-order executing and has limited hit-under-miss leading to limited memory latency.