Lab 1

ECSE 444

Isbat Islam
261117219

Neil Joe George
261107755

***Abstract*—This report explores the implementation and performance evaluation of a Kalman filter using assembly, plain C, and CMSIS-DSP on an ARM Cortex M4 processor. Key performance metrics, including execution time and statistical properties of filter output, are analyzed.**

***Index Terms*—Kalman Filter, ARM Cortex M4, Assembly, C, CMSIS-DSP, Performance Profiling**

I. Implementation Code

This section presents the code snippets used for each part of the implementation. These snippets illustrate the Kalman filter implementation in both standard C and CMSIS-DSP, as well as the core functions involved in the project.

Figure 1 shows the main implementation file that sets up the Kalman filter states and processes the test array data. The measurement values are passed through different Kalman filter implementations for comparison.

Figure 2 contains the functions responsible for processing the test data, performing basic mathematical operations and preparing the data for filtering.

Figure 3 presents the CMSIS-DSP optimized Kalman filter implementation, which leverages ARM's DSP library functions for efficient computation.

Figure 4 shows the standard C implementation of the Kalman filter, providing a baseline comparison for the DSP-optimized version.

Figure 5 presents the assembly code used for array processing in the implementation.

Figure 6 illustrates the core assembly code implementation for the Kalman filter.

```
int main(void) {
  /* USER CODE BEGIN 2 */
  float TEST_ARRAY[] = {
      10.4915760032, 10.1349974709, 9.539925918229, 9.603118787706, 10.48588917931, 10.1104642352,
      9.510666931906, 9.7575565644$, 9.821540782$, 10.29065641931, 10.48613286$1, 9.57321181356,
      9.70882714139, 10.4359069357, 9.70644021369, 10.2709894039, 10.0823149505, 10.2954563443,
      9.57130449017, 9.66832136478, 10.4521677502, 10.4287240667, 10.1833650752, 10.0066049721,
      10.3279461634, 10.4767210803, 10.3790964606, 10.1937408814, 10.0318963522, 10.4939180917,
      10.2381858895, 9.59703103028, 9.62757986516, 10.1816981174, 9.657037731$8, 10.3905666598,
      10.0941977598, 9.93515274393, 9.71017053437, 10.0303874255, 10.0173504197, 9.69022731474,
      9.73902896102, 9.52524419732, 10.3270730526, 9.54695650657, 10.3573960542, 9.88773266876,
      10.1685038683, 10.1683694089, 9.88406620159, 10.3290065898, 10.2547227265, 10.4733422906,
      10.0133952458, 10.4205695583, 9.71335255372, 9.89061396699, 10.1652744131, 10.2580948608,
      10.3465431058, 9.98446410493, 9.79376005657, 10.202518901, 9.83867150985, 9.89532966869,
      10.2885062658, 9.97748768804, 10.0403923759, 10.1538911808, 9.78303667556, 9.72420149909,
      9.59117495073, 10.1716116012, 10.2015818969, 9.90650056596, 10.3251329834, 10.4550120431,
      10.4925749165, 10.1548177178, 9.60547133785, 10.4644672766, 10.2326496615, 10.2279703226,
      10.3535284606, 10.2437410625, 10.3851531317, 9.90784804928, 9.98208344925, 9.52778805725,
      9.69323876912, 9.92987312087, 9.7393892520$, 9.60543743477, 9.79600805462, 10.4950988486,
      10.2814361401, 9.7985283333,  9.6287888922,  10.449153899$, 9.5799256668};

  // Initialize filter with more conservative values
  kalman_state kState = {1.0f, 0.1f, 5.0f, 1.0f, 0.0f};  // Higher process and measurement noise
  kalman_state CKState= {1.0f, 0.1f, 5.0f, 1.0f, 0.0f};  // Higher process and measurement noise
  kalman_state CMSISKState= {1.0f, 0.1f, 5.0f, 1.0f, 0.0f};  // Higher process and measurement noise
  DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;  // Enable the cycle counter

  int measurementCount = 101;
  float result[101];
  float resultC[101];
  float resultCMSIS[101];

  uint32_t startTime, endTime;
  int asmNbCycles = 0, CNbCycles = 0, CMSISNbCycles = 0;
  int errCode;

  // Asm Implementation
  for (int i = 0; i < measurementCount; i++) {
      startTime = DWT->CYCCNT;  // Start Timer
    errCode = KalmanFilter(&kState, TEST_ARRAY[i]);
    endTime = DWT->CYCCNT; // End Timer
    asmNbCycles += (endTime - startTime);
    if (errCode != 0)
      return errCode;
    // result array (output array) is completed here. No need to populate it again later.
    result[i] = kState.x;
  }

  // C Implementation
  for (int i = 0; i < measurementCount; i++) {
    startTime = DWT->CYCCNT;
    errCode = KalmanFilter_C(&CKState, TEST_ARRAY[i]);
    endTime = DWT->CYCCNT;
    CNbCycles += (endTime - startTime);
    if (errCode != 0)
      return errCode;
    resultC[i] = CKState.x;
  }

  // CMSIS Implementation
  for (int i = 0; i < measurementCount; i++) {
    startTime = DWT->CYCCNT;
    errCode = KalmanFilter_C_CMSIS(&CMSISKState, TEST_ARRAY[i]);
    endTime = DWT->CYCCNT;
    CMSISNbCycles += (endTime - startTime);
    if (errCode != 0)
      return errCode;
    resultCMSIS[i] = CMSISKState.x;

  }
  /* USER CODE END 2 */
}

/* USER CODE BEGIN 4 */
int Kalmanfilter(float* InputArray, float* OutputArray, kalman_state* kstate, int Length) {
  for (int i = 0; i < Length; i++) {
    int err_code;
    err_code = KalmanFilter_C(kstate, InputArray[i]);
    err_code = KalmanFilter_C_CMSIS(kstate, InputArray[i]);
    err_code = KalmanFilter(kstate, InputArray[i]);
    if (err_code != 0) {
      return err_code;
    }
    OutputArray[i] = kstate->x;
  }

  return 0;
}

/* USER CODE END 4 */
```

Fig. 1. main.c file

```c
#include "analysis.h"

void CMSIS_analysis(float* TEST_ARRAY, float* result, int measurementCount){
    float difference[101];
    float mean, stdDev;
    float correlation[2 * measurementCount - 1];
    float convolution[2 * measurementCount - 1];

    // Difference
    arm_sub_f32(TEST_ARRAY, result, difference, measurementCount);

    // Mean and Standard Deviation
    arm_mean_f32(difference, measurementCount, &mean);
    arm_std_f32(difference, measurementCount, &stdDev);

    // Correlation
    arm_correlate_f32(TEST_ARRAY, measurementCount, result, measurementCount, correlation);

    // Convolution
    arm_conv_f32(TEST_ARRAY, measurementCount, result, measurementCount, convolution);
}

void C_analysis(float* TEST_ARRAY, float* result, int measurementCount){
    float difference[101];
    float mean = 0.0f, stdDev = 0.0f;
    float correlation[2 * measurementCount - 1];
    float convolution[2 * measurementCount - 1];

    // Difference
    for (int i = 0; i < measurementCount; i++) {
        difference[i] = TEST_ARRAY[i] - result[i];
    }

    // Mean
    for (int i = 0; i < measurementCount; i++) {
        mean += difference[i];
    }
    mean /= measurementCount;

    // Standard Deviation
    for (int i = 0; i < measurementCount; i++) {
        stdDev += (difference[i] - mean) * (difference[i] - mean);
    }
    stdDev = sqrtf(stdDev / measurementCount);

    // Correlation
    for (int lag = 0; lag < 2 * measurementCount - 1; lag++) {
        correlation[lag] = 0.0f;
        for (int i = 0; i < measurementCount; i++) {
            int j = lag + i - measurementCount + 1;
            if (j >= 0 && j < measurementCount) {
                correlation[lag] += TEST_ARRAY[i] * result[j];
            }
        }
    }

    // Convolution
    for (int i = 0; i < 2 * measurementCount - 1; i++) {
        convolution[i] = 0.0f;
        for (int j = 0; j < measurementCount; j++) {
            if (i >= j && (i - j) < measurementCount) {
                convolution[i] += TEST_ARRAY[j] * result[i - j];
            }
        }
    }
}
```

Fig. 2. Analysis functions used in the implementation

```c
#include "KalmanFilter_C_CMSIS.h"

int KalmanFilter_C_CMSIS(kalman_state* kState, float measurement) {
    float32_t temp1[1], temp2[1], one[1] = {1.0f};

    // P = P + Q
    arm_add_f32(&(kState->p), &(kState->q), &(kState->p), 1);
    if (isinf(kState->p) || isnan(kState->p)) {
        return KALMAN_OVERFLOW; // Overflow error
    }
    if ((kState->p + kState->r) < FLT_EPSILON){
        return KALMAN_DIV_BY_ZERO;
    }

    // temp1 = P + R
    arm_add_f32(&(kState->p), &(kState->r), temp1, 1);

    // K = P / temp1
    kState->k = kState->p / temp1[0];

    if (isinf(kState->k) || isnan(kState->k)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    // temp1 = measurement - X
    arm_sub_f32(&measurement, &(kState->x), temp1, 1);

    // temp2 = K * temp1
    arm_mult_f32(&(kState->k), temp1, temp2, 1);

    // X = X + temp2
    arm_add_f32(&(kState->x), temp2, &(kState->x), 1);
    if (isinf(kState->x) || isnan(kState->x)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    // temp1 = 1 - K
    arm_sub_f32(one, &(kState->k), temp1, 1);

    // P = temp1 * P
    arm_mult_f32(temp1, &(kState->p), &(kState->p), 1);

    if (isinf(kState->p) || isnan(kState->p)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    return KALMAN_SUCCESS;
}
```

Fig. 3. CMSIS-DSP Kalman filter implementation

```c
#include "KalmanFilter_C.h"

int KalmanFilter_C (kalman_state* kState, float measurement){

    kState->p = kState->p + kState->q;
    if (isinf(kState->p) || isnan(kState->p)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    float denominator = kState->p + kState->r;
    if (denominator < FLT_EPSILON) {
        return KALMAN_DIV_BY_ZERO;
    }

    // Update: Calculate the Kalman gain
    kState->k = kState->p / denominator;


    if (isinf(kState->k) || isnan(kState->k)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    kState->x += kState->k * (measurement - kState->x);
    if (isinf(kState->x) || isnan(kState->x)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    kState->p *= (1.0f-kState->k);
    if (isinf(kState->p) || isnan(kState->p)) {
        return KALMAN_OVERFLOW; // Overflow error
    }

    return KALMAN_SUCCESS;
}
```

Fig. 4. Standard C Kalman filter implementation

```
.syntax unified
.align 16

.global KalmanFilter

KalmanFilterArray:
    push {r5-r6}
    vpush {s16-s21}

    // r0 kalman state addr
    // r1 measurement arr addr
    // r2 result arr addr

    // s0 contains measurement
    vldr s16, [r0, #0]      // q
    vldr s17, [r0, #4]      // r
    vldr s18, [r0, #8]      // x
    vldr s19, [r0, #12]     // p
    vldr s20, [r0, #16]     // k
    vsub.f32 s21, s21, s21  // -> 0.0
    vldr s22, [r1]
    mov r5, #101            // size of measurement array
    mov r6, #0              // number of iteration done


iteration:
    vadd.f32 s19, s19, s16  // self.p = self.p + self.q
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vadd.f32 s20, s19, s17  // self.k = self.p + self.r
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    // handle zero div error (see epsilon method)
    vcmp.f32 s20, #0
    vmrs APSR_nzcv, FPSCR
    beq div_by_zero_handler


    vdiv.f32 s20, s19, s20 // self.k = selp.p / self.k
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vsub.f32 s21, s22, s18  // y = measurement - self.x
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vmla.f32 s18, s20, s21 // self.x = self.x + self.k * y
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    // store the x value inside of result array
    vstr s18, [r2]
    // increment pointer by 4
    add r2, r2, #4

    add r1, r1, #4
    vldr s22, [r1]

    vmov.f32 s21, #1.0 // y = 1 check if load 1 works
    vsub.f32 s21, s21, s20 // y = 1 - self.k
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vmul.f32 s19, s21, s19 // self.p = y * self.p
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    add r6, r6, #1
    cmp r5, r6
    BNE iteration


    vstr s16, [r0]      // q
    vstr s17, [r0, #4]  // r
    vstr s18, [r0, #8]  // x
    vstr s19, [r0, #12] // p
    vstr s20, [r0, #16] // k
    mov r0, #0          // If success, return code 0
    B return

div_by_zero_handler:
    mov r0, #1
    B return

overflow_handler:
    mov r0, #2
    B return

return:
    pop {r5-r6}
    vpop {s16-s21}
    bx lr // return to C
```

Fig. 5. Array assembly code implementation

```
.syntax unified
.align 16

.global KalmanFilter

KalmanFilter:
    push {lr}
    vpush {s16-s21}
    // s0 contains measurement
    vldr s16, [r0, #0]      // q
    vldr s17, [r0, #4]      // r
    vldr s18, [r0, #8]      // x
    vldr s19, [r0, #12]     // p
    vldr s20, [r0, #16]     // k
    vsub.f32 s21, s21, s21  // -> 0.0

    vadd.f32 s19, s19, s16  // self.p = self.p + self.q
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vadd.f32 s20, s19, s17  // self.k = self.p + self.r
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    // handle zero div error (see epsilon method)
    vcmp.f32 s20, #0
    vmrs APSR_nzcv, FPSCR
    beq div_by_zero_handler


    vdiv.f32 s20, s19, s20 // self.k = selp.p / self.k
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vsub.f32 s21, s0, s18  // y = measurement - self.x
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vmla.f32 s18, s20, s21 // self.x = self.x + self.k * y
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vmov.f32 s21, #1.0 // y = 1 check if load 1 works
    vsub.f32 s21, s21, s20 // y = 1 - self.k
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vmul.f32 s19, s21, s19 // self.p = y * self.p
    vmrs r1, FPSCR
    tst r1, #0b1111
    bne overflow_handler

    vstr s16, [r0]      // q
    vstr s17, [r0, #4]  // r
    vstr s18, [r0, #8]  // x
    vstr s19, [r0, #12] // p
    vstr s20, [r0, #16] // k
    mov r0, #0          // If success, return code 0
    B return

div_by_zero_handler:
    mov r0, #1
    B return

overflow_handler:
    mov r0, #2
    B return

return:
    vpop {s16-s21}
    bx lr // return to C
```

Fig. 6. Assembly code implementation

II. Result

A. Run the program and record the execution for a trace of your choice that nicely illustrates the properties of the Kalman filter, such as the convergence towards the input stream and the statistical properties of the deviation to the input.

In this chosen trace example, the measurement input stream is always 100. After trial and error, fiddling with the Kalman Filter parameters, it was determined that $q = 0.01$, $r = 5.0$, $x = 100.0$, $p = 100.0$, and $q = 0.0$ attenuates the noise on the Gaussian Noise Measurement signal (generated using Python NumPy) considerably. The output is shown in Figure 7.

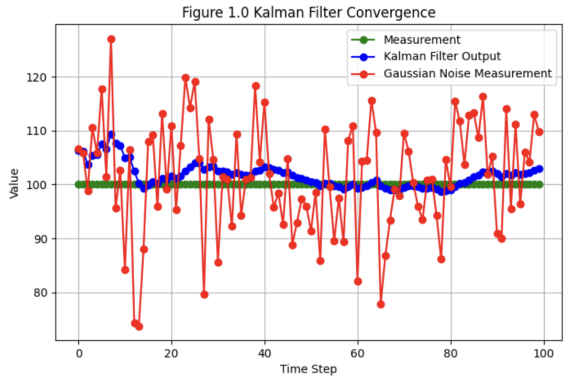

Fig. 7. Kalman filter output demonstrating convergence.

Figure 7 clearly shows that the output of the Kalman filter (blue line) converges towards the input stream of measurement (green line), despite random fluctuations of the noisy signal (red line). These data points for the blue trace were obtained after running the KalmanFilterAssembly implementation and exporting the data points to Python to plot the graph.

B. Explicitly calculate in your program the difference to the input stream, standard deviation and average of that difference, as well as the correlation and convolution between the two streams.

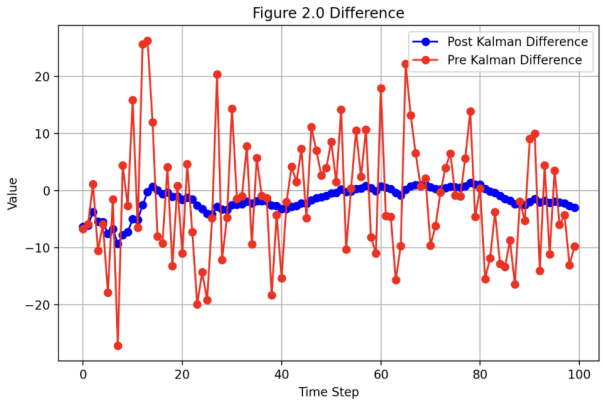

Fig. 8. Error statistics visualization.

The standard deviation and the average of the post-Kalman difference are 10.02 and -0.205, respectively.

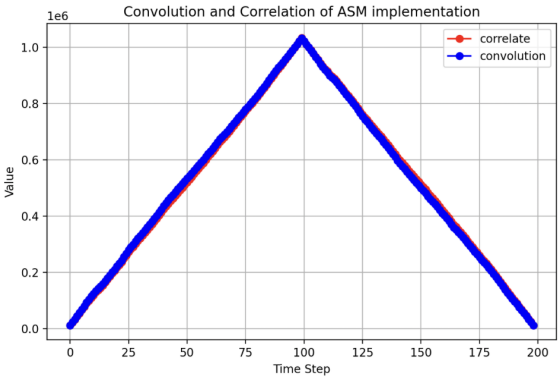

Fig. 9. Error statistics computed using CMSIS-DSP.

C. Using CMSIS-DSP, perform the same functions as in 2.

The average and standard deviation are -0.206 and 10.019, respectively.

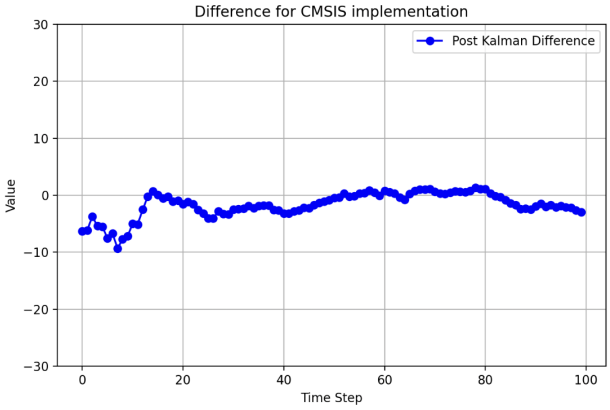

Fig. 10. CMSIS-DSP computed statistics.

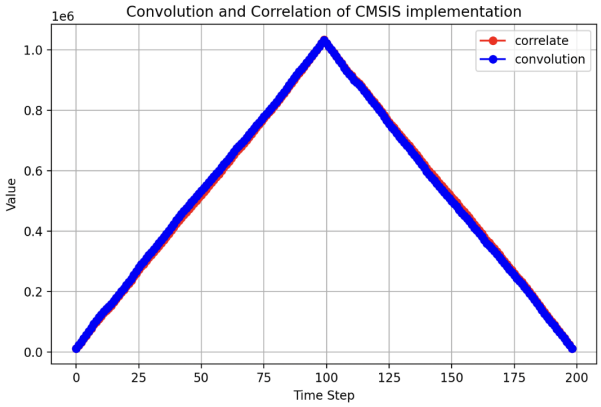

Fig. 11. Execution time comparison of implementations.

D. Using code profiling, report the running times for all the procedures with your code and CMSIS-DSP. You should get the results as fast as possible.

For code profiling and to measure the speed of the implementation, we are using the number of cycles the program takes to execute and go over the whole TEST_ARRAY. In the case of CMSIS-DSP, the number of cycles it took is 68904 cycles. At 120MHz, that turns out to be 0.57ms. For the assembly implementation, it took 12594 cycles, or around 0.1ms.

E. Rewrite your core Kalman filter code in plain C as well as in C using the CMSIS-DSP. Keep in mind that it is possible to obtain the "scalar" version of CMSIS-DSP when needed.

Yes, we can do scalar CMSIS-DSP when we enter a vector size of 1; by doing so, it's a scalar value even though it's vector.

F. Report the profiling data that compares the three versions of Kalman filter core (assembly, plain C and CMSIS-DSP).

As observed in the table above, C requires 26,190 cycles, CMSIS requires 68,570, and assembly requires 12,594 cycles, which amounts to 0.22ms, 0.57ms, and 0.1ms, respectively.

Fig. 12. Comparison of execution cycles for each implementation.

G. Summarize the lessons learned on the use of assembler, C, and CMSIS-DSP by selecting the suitable profiling data.

The results obtained suggest that, for our situation, the assembly implementation was the fastest by a long shot, around 2x faster than the C implementation, and around 6x faster than the CMSIS implementation. Thus, if speed is our only concern, the assembly implementation would be the most suitable option.

However, between C and CMSIS, we would expect that CMSIS would be faster than C, but the results show otherwise. Indeed, CMSIS assumes vectorized data, and for our 1D Kalman Filter, there was no such data. Thus, the overhead caused by CMSIS slowed the performance down substantially, leaving C to be faster in this situation.

H. Finally, run the code on the actual processor and use the debugger to inspect and modify the program variables while the code is running. Summarize in the report on the rules: which variables can be watched, modified, and whether and how that can be done without stopping the processor.

The variables that can be modified are everything inside the general-purpose registers, without stopping the processor (while a breakpoint is reached and it is paused, without stopping the debugging session).

However, special registers like the PC and the SP cannot be modified, as even if it allows you to change the value, when running the program, it crashes and sends you to the MemManage_Handler. The FPU registers cannot be modified, as even though the program does not crash, the program halts and is unresponsive, while displaying the following error:

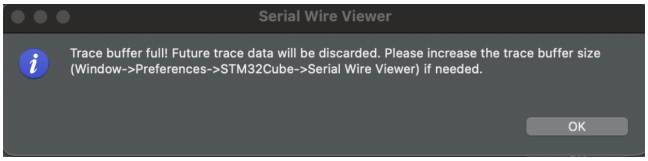

Fig. 13. Debugger error when modifying restricted variables.