Kalman Filter using Floating-Point Assembly and its Evaluation with C and CMSIS-DSP

Simon Li
dept. of electrical, computer and software engineering
Montreal, Canada
xi.yang.li@mail.mcgill.ca

Yue Lang Li
dept. of electrical, computer and software engineering
Montreal, Canada
yue.l.li@mail.mcgill.ca

***Abstract*—This report includes the program implementation method in STM32CubeIDE and the experimental results of the Kalman filter using floating-point Assembly language and C. The first section will be mainly an Assembly code subroutine that tracts the value of the Kalman filter for single-variable, and the second section will be to rewrite the filter in plain C and in CMSIS-DSP programs to perform the subroutine repeatedly for an array of inputs. It also includes properties of the Kalman filter such as its convergence towards the input stream and the statistical properties of the deviation from the input.**

***Index Terms*—Kalman filter, STM32CubeIDE, ARM Assembly, plain C, CMSIS-DSP**

I. Introduction

The Kalman filter is an important algorithm in the field of signal processing that provides the estimation for the state value of a system using the parameter's noise variances. This recursive algorithm provides an optimal solution to the estimation problem, in the sense that the mean and covariance of the estimate are optimal, given the measurements and the underlying system dynamics. Significantly, the algorithm is widely used in fields such as control, navigation, and signal processing, where it is crucial to estimate the state of a system based on noisy and limited observations.

This laboratory will compare the experimental data (e.g. speed) of three different codes implementation using different languages, and the properties of the filter tracking the input variable such as the standard deviation, the correlation of the original and filter, and their convolution.

II. Implementation

A. ARM Assembly

First, the Kalman filter was implemented in assembly. The register R0 contains a pointer to the input measurements array, R1 a pointer to the output array, R2 a pointer to the Kalman state and R3 contains the length. To switch from a single variable to an array of values, a loop was implemented to repeatedly calculate the output values for each iteration. During each loop, the values in the Kalman state (i.e. p, k, x) are updated according to the Python code definition of update, and the values of R0 and R1 are incremented by 4 to point to the next value in the array. At each iteration, division by zero, overflow, and underflow are checked by moving the values of the floating point status register (FPSCR) to the application program status register (APSR) to account for edge cases. In cases of error, we branch to the error handler where an error code of -1 is returned. At the end of each iteration, the APSR flag bits are cleared.

B. Plain C

As for the plain C code, the same operations as in ARM assembly were implemented. The function has the same signature as the assembly one, and during each iteration, we update the prediction error covariance, calculate the Kalman gain, update the estimate of the state, and finally write and update the estimate of the state (x) to the output array. To account for edge cases, we check if the output value is NaN or INF and return -1 if either is true.

C. CMSIS-DSP

Finally, we re-implemented the algorithm once again but using the DSP library this time. The arm math library contains in-built functions that are optimized for vector and floating-point operations. In our implementation of the Kalman filter, we used the functions arm_add_f32, arm_sub_f32, and arm_mult_f32. Interestingly, no division functions were found in the library. Therefore we used the normal division operation in C to calculate and update the prediction error covariance. This could have a significant effect on the performance of the DSP implementation. In the next part, the performance of the 3 implementations will be analyzed and compared.

III. Performance

First, we used the Instrumentation Trace Microcell (ITM), which is designed to add timestamps to trace events. If we create a timestamp before and after the function call, we can approximate the execution time of the Kalman filter function. We believe that ITM_Port32(n) is a location in memory and that setting it to a value will generate a trace packet with that value as the data. This also generates a timestamp in terms of elapsed cycle count and wall-clock time. We executed 101 calls to the function using the provided input measurement array with 101 values and recorded a time of 0.0648 ms per call. Then, the Kalman filter was run and a trace was recorded to illustrate its properties, specifically the convergence towards the input stream and the statistical properties of the deviation from the input. The recorded trace showed a decrease in deviation from the input over time, demonstrating the effectiveness of the Kalman filter in reducing error. Subsequently, the difference between the input stream and the output from the Kalman filter was explicitly calculated. The results showed a standard deviation of 10.069 and an average of -0.025. The correlation and convolution between the two streams were also calculated and further confirmed the accuracy of the Kalman filter in tracking the input stream. The core Kalman filter code was rewritten in plain C. It is worth mentioning that this implementation is 262 % slower than the assembly implementation, with an execution time of 0.1699 ms compared to 0.0648 ms for the assembly implementation. Using CMSIS-DSP, the same functions performed in 2 were calculated. Using code profiling, the running times for all the procedures with our code and CMSIS-DSP were reported, with running time measured at 0.3126 ms. Even though the Kalman filter run-time is longer for CMSIS-DSP, the convolution time will be shorter than the previous implementation.

IV. Results

From the previous section, we have recorded the run-time for the Kalman filter for the three diffrents implementations. To make the results clearer, we have included the Kalman filter run-time table below.

TABLE I
Kalman filter run-time of different implementations

Kalman filter run time (ms)		
ARM assembly	plain C	CMSIS-DSP
0.064775	0.169900	0.312642

From this table, we can see that the fastest run-time is for the ARM assembly implementation, as the Kalman filter only relies on atomic operations. The plain C implementation is at second place, with DSP being the slowest.

TABLE II
Standard deviation, correlation, and convolution run-time of different implementations

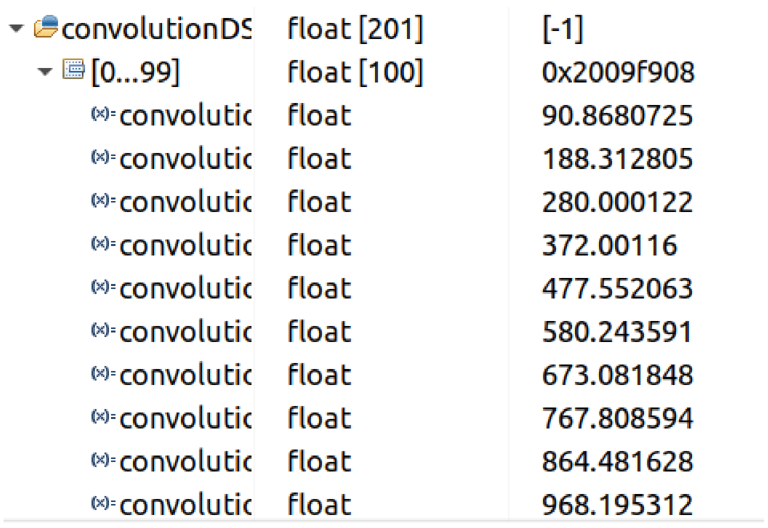
Statistics timing (μ s)	Standard dev	Correlation	Convolution
Plain c	1094.5	4113	4114
DSP	7.442	585.1	585.4

From the obtained results, we can see that DSP is faster for convolution but slower for the Kalman as mentioned in the previous section. The reason for this is that the DSP implementation is optimized for vector computation and both convolution and correlation, and uses large arrays. However, since the Kalman filter does computations on float scalars, then it is clear that DSP is not very effective for those computations.

V. Conclusion

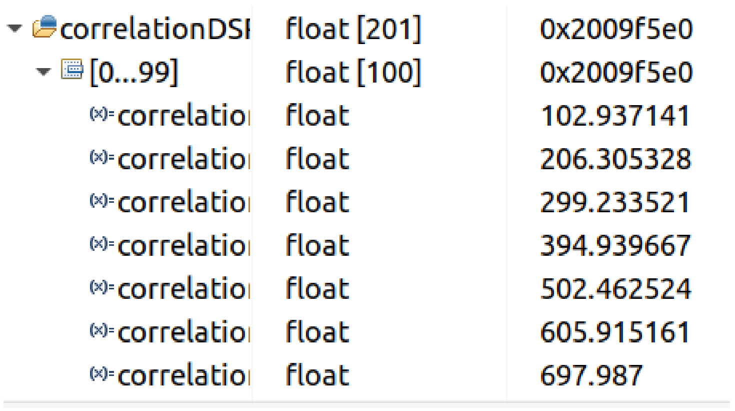
After analyzing the profiling data, 0.0648 ms was observed for the Kalman filter running time using assembly, 0.1699 ms using plain C and 0.3126 ms using CMSIS-DSP. Based on these results, it can be concluded that CMSIS-DSP is the most efficient option for implementing the Kalman filter code. However, it is important to consider the scalar version of CMSIS-DSP when necessary. The code was run on the actual processor and the debugger was used to inspect and modify the program variables. It was found that all of the five variables (q, r, x, p, and k) can be watched and modified. These variables can be modified without stopping the processor by simply entering the new data value without the need to stop the program. Overall, the use of the debugger allowed for more efficient debugging and development of the Kalman filter code. In conclusion, the experiments performed aimed at exploring the properties and performance of Kalman filter. The results showed the convergence of the filter towards the input stream and the statistical properties of the deviation to the input. The difference in the input stream, standard deviation, average, correlation, and convolution between the two streams was calculated both through a custom program and CMSIS-DSP. The code profiling results indicated a faster performance when using CMSIS-DSP. The core Kalman filter code was rewritten in plain C and in C using CMSIS-DSP and the profiling data compared the three versions of the filter core. The lessons learned from the use of assembler, C, and CMSIS-DSP were summarized by selecting the suitable profiling data. Finally, the code was run on an actual processor and the debugger was used to inspect and modify the program variables without stopping the processor. The results obtained from the experiments provide valuable insights into the properties and performance of Kalman filter. The faster performance when using CMSIS-DSP highlights the significance of libraries and tools in improving the efficiency of code. The ability to modify program variables while the code is running without stopping the processor is also a significant finding. In the future, these results could be used as a reference for further research and development in the field of Kalman filter and their applications in fields such as autonomous driving. The lessons learned on the use of assembler, C, and CMSIS-DSP can also be applied to other projects, leading to more efficient and optimized code. Overall, we believe that the research has significant implications for the application of Kalman filter in various fields.

VI. Appendixes



convolutionDS	float [201]	[-1]
[0...99]	float [100]	0x2009f908
convolutic	float	90.8680725
convolutic	float	188.312805
convolutic	float	280.000122
convolutic	float	372.00116
convolutic	float	477.552063
convolutic	float	580.243591
convolutic	float	673.081848
convolutic	float	767.808594
convolutic	float	864.481628
convolutic	float	968.195312

Fig. 1. Convolution results for ARM assembly



correlationDSI	float [201]	0x2009f5e0
[0...99]	float [100]	0x2009f5e0
correlatio	float	102.937141
correlatio	float	206.305328
correlatio	float	299.233521
correlatio	float	394.939667
correlatio	float	502.462524
correlatio	float	605.915161
correlatio	float	697.987

Fig. 2. Correlation results for ARM assembly