EE5332 PROJECT UPDATE-2 **ARUNKRISHNA A M S (EE19B001)**
KALMAN FILTER OPTIMIZATIONS **May 5, 2022**

Following were done and explored in the last two weeks:

- A working C++ code implementation of Kalman filter which uses “Eigen” matrix library was completed. The inputs were a 6-state variable vector, 2 dimensional measurement vector.
  (**File**)

The above code is not synthesizable using Vivado HLS. The “Eigen” library is optimized for large matrices – but has huge overhead. Since the input example matrices that we considered were of lower dimensions, the overhead was way too much for the library to be useful.

- Hence I (& Surya) implemented C functions for matrix inverse (using Gauss Jordan Elimination) and matrix multiplication. With the C functions the kalman filter was implemented. (**File**)

On both the implementations, I tried out a couple of profiling tools to profile the individual elemental functions and find the bottleneck (out of 6 equations, which equation is the bottleneck and which runs the fastest. Within the equation what operation causes it to be extremely slow?)

While profiling I ensured that the functions are not made inline.

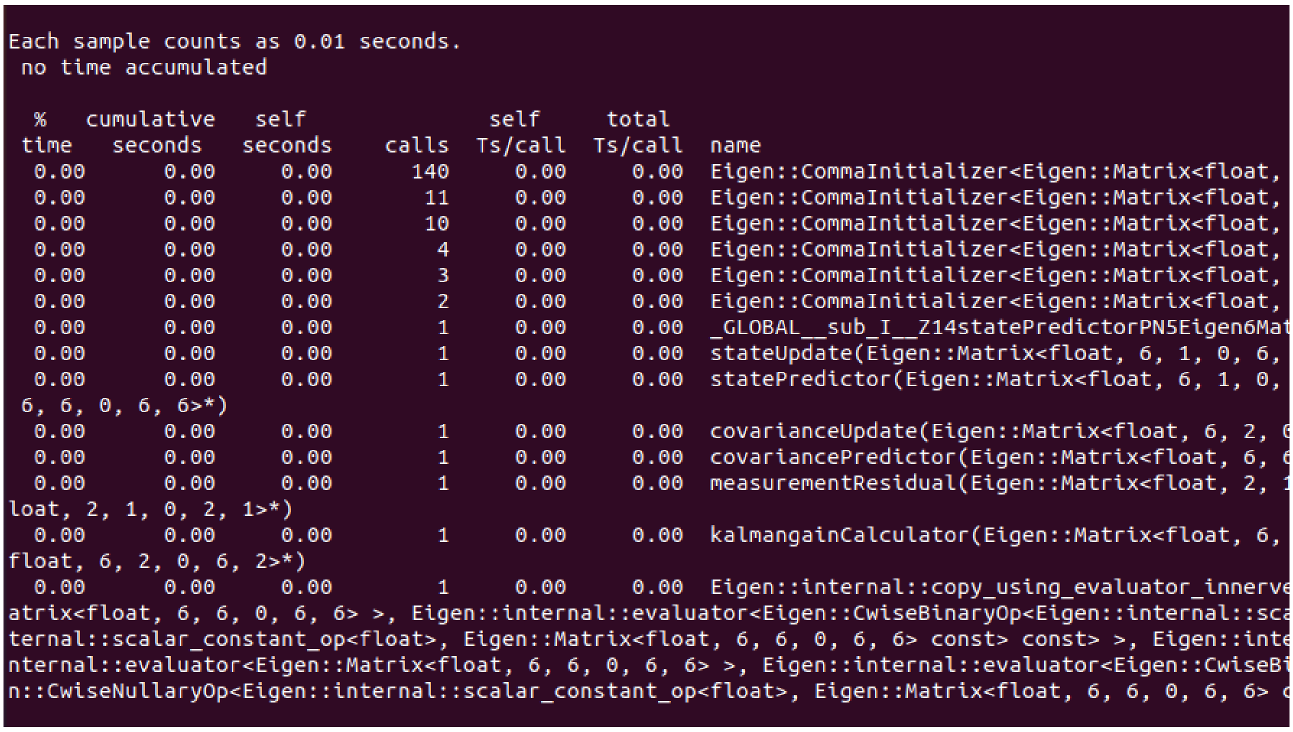
Following profiling tools were tried out:
1. gprof
2. Valgrind-Callgrind
3. google-performance tools

gprof profiling tool

gprof is a sampling based profiling tool. Since it is a sampling based profiling tool – it may not be that accurate but at the same time, sampling and instrumentation are quite fast – hence doesn’t slow down the execution. **gprof2dot** was used to then generate a call-graph.

The sampling frequency of the tool is 0.01 seconds or 10 milliseconds.

For both the huge “Eigen overhead” and each of the function corresponding to the 6 equations of kalman filter:
- stateUpdate()
- statePredictor()
- covarianceUpdate()
- covariancePredictor()
- measurementResidual()
- kalmangainCalculator()



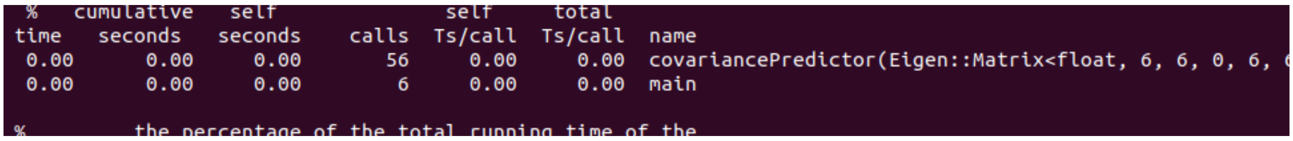
```
Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      140     0.00     0.00  Eigen::CommaInitializer<Eigen::Matrix<float,
  0.00      0.00     0.00       11     0.00     0.00  Eigen::CommaInitializer<Eigen::Matrix<float,
  0.00      0.00     0.00       10     0.00     0.00  Eigen::CommaInitializer<Eigen::Matrix<float,
  0.00      0.00     0.00        4     0.00     0.00  Eigen::CommaInitializer<Eigen::Matrix<float,
  0.00      0.00     0.00        3     0.00     0.00  Eigen::CommaInitializer<Eigen::Matrix<float,
  0.00      0.00     0.00        2     0.00     0.00  Eigen::CommaInitializer<Eigen::Matrix<float,
  0.00      0.00     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z14statePredictorPN5Eigen6Mat
  0.00      0.00     0.00        1     0.00     0.00  stateUpdate(Eigen::Matrix<float, 6, 1, 0, 6,
  0.00      0.00     0.00        1     0.00     0.00  statePredictor(Eigen::Matrix<float, 6, 1, 0,
 6, 6, 0, 6, 6>*)
  0.00      0.00     0.00        1     0.00     0.00  covarianceUpdate(Eigen::Matrix<float, 6, 2, 6
  0.00      0.00     0.00        1     0.00     0.00  covariancePredictor(Eigen::Matrix<float, 6, 6
  0.00      0.00     0.00        1     0.00     0.00  measurementResidual(Eigen::Matrix<float, 2, 1
loat, 2, 1, 0, 2, 1>*)
  0.00      0.00     0.00        1     0.00     0.00  kalmangainCalculator(Eigen::Matrix<float, 6,
float, 6, 2, 0, 6, 2>*)
  0.00      0.00     0.00        1     0.00     0.00  Eigen::internal::copy_using_evaluator_innerve
atrix<float, 6, 6, 0, 6, 6> >, Eigen::internal::evaluator<Eigen::CwiseBinaryOp<Eigen::internal::sca
ternal::scalar_constant_op<float>, Eigen::Matrix<float, 6, 6, 0, 6, 6> const> const> >, Eigen::inte
nternal::evaluator<Eigen::Matrix<float, 6, 6, 0, 6, 6> >, Eigen::internal::evaluator<Eigen::CwiseBi
n::CwiseNullaryOp<Eigen::internal::scalar_constant_op<float>, Eigen::Matrix<float, 6, 6, 0, 6, 6> c
```

The average number of time spent in each of the functions mentioned above is 0.00 milliseconds (based on self and total Ts/call). This means that with the given sampling frequency of 100 Hz, we are not able to profile the individual functions.

The profiling output is converted to a graph using gprof2dot (**File attached**)
It shows up as 100% for each function - since each of the functions run so fast that everything is done within 0.01 seconds.

Note: Each of the above functions of the kalman filter is called only once (single call).

When fully optimized using -o3 flag, we observe the following

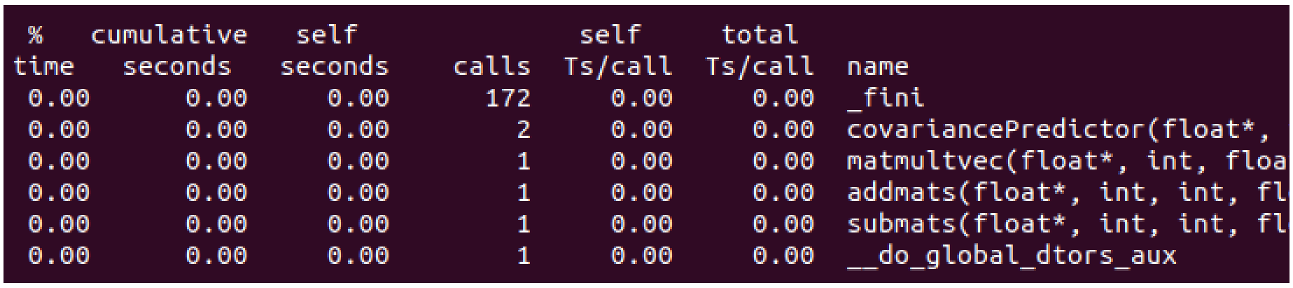


```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00       56     0.00     0.00  covariancePredictor(Eigen::Matrix<float, 6, 6, 0, 6, 6
  0.00      0.00     0.00        6     0.00     0.00  main

 %          the percentage of the total running time of the
```

Only main and covariancePredictor shows up. The overhead disappears, while covariancePredictor has 50+ calls compared to 1 call it had during -o1 optimization. I couldn’t understand why this happens ?

As expected both the functions is run so fast within 0.01 seconds that gprof wasn’t able to sample anything.
The profiling output in graph format for the same (**File attached**)

We try to do the analysis for our C code (matrix operations implemented by us) using -o1 optimization. Again as expected the functions and the entire program ran so fast that they were not sampled.


```
 %   cumulative   self              self     total           
time   seconds   seconds    calls  Ts/call  Ts/call  name    
 0.00      0.00     0.00      172     0.00     0.00  _fini
 0.00      0.00     0.00        2     0.00     0.00  covariancePredictor(float*,
 0.00      0.00     0.00        1     0.00     0.00  matmultvec(float*, int, floa
 0.00      0.00     0.00        1     0.00     0.00  addmats(float*, int, int, fl
 0.00      0.00     0.00        1     0.00     0.00  submats(float*, int, int, fl
 0.00      0.00     0.00        1     0.00     0.00  __do_global_dtors_aux
```

The output graph is **attached.**

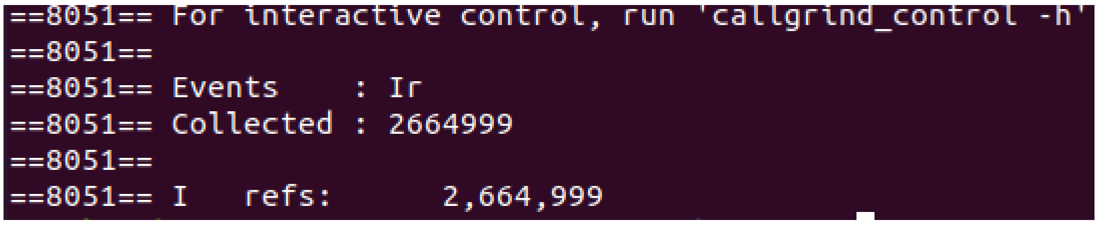
Overall we find that because of the low sampling frequency of gprof and very high execution rates of the individual functions, no useful information was obtained. We couldn’t find an easy way to change the sampling frequency – hence we abandoned this profiler.

Valgrind – Callgrind

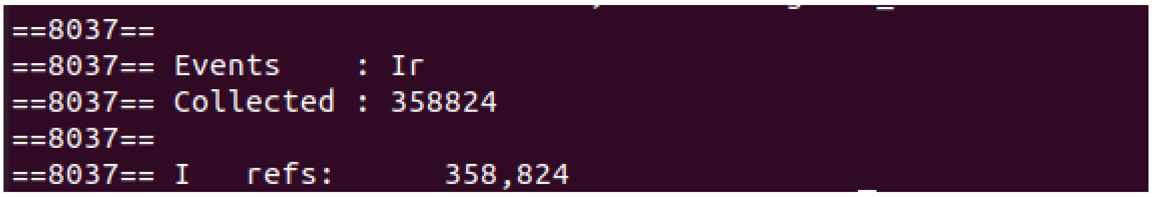
Valgrind runs the program in **Valgrind** virtual machine. **Callgrind** – Valgrind’s tool is used to profile the code. Since Valgrind is a simulated virtual machine, it makes profiling very accurate but at the same time, it makes execution extremely slow(can be as slow as 50x times).

Callgrind’s output can be visualized through **Kcachegrind.**

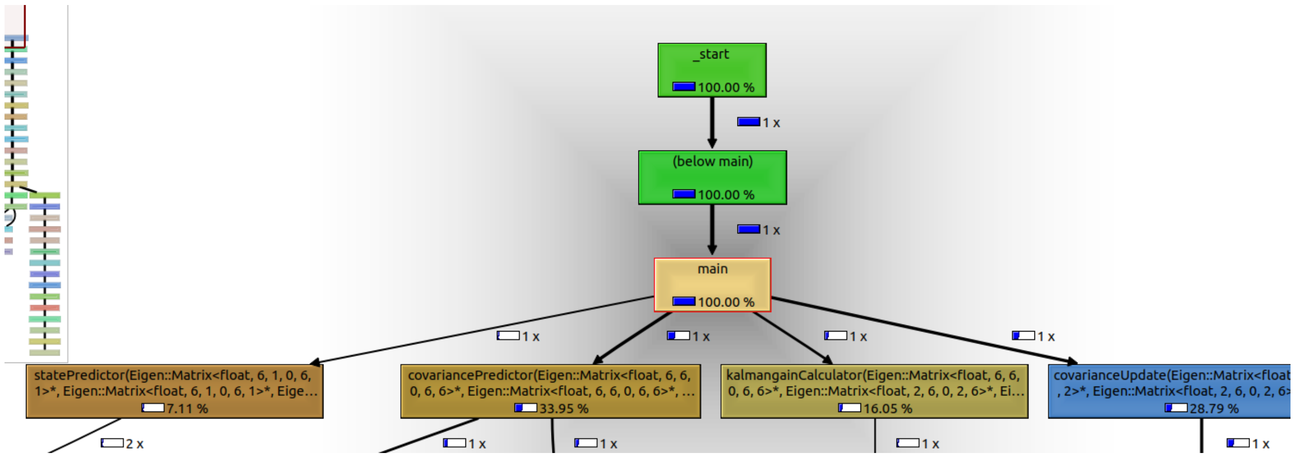
When we try to profile the C++ code involving the Eigen library with -o optimization, we observe the following:


```
==8051== For interactive control, run 'callgrind_control -h'
==8051==
==8051== Events    : Ir
==8051== Collected : 2664999
==8051==
==8051== I   refs:      2,664,999
```

The total number of instructions is around 2.5 millions. Since the overhead is unavoidable, we try to profile from the “main()” function. If we do the same, we observe that the instructions are reduced to just 350000+ instructions.


```
==8037==
==8037== Events    : Ir
==8037== Collected : 358824
==8037==
==8037== I   refs:      358,824
```

The profiling output for the last case (profiling only main function and not overhead) is viewed through kcachegrind. The **Attached file** talks about the Call-graph and the time spent in individual functions


_start
100.00 %
1 x
(below main)
100.00 %
1 x
main
100.00 %
1 x
1 x
1 x
1 x
statePredictor(Eigen::Matrix<float, 6, 1, 0, 6, 1>*, Eigen::Matrix<float, 6, 1, 0, 6, 1>*, Eige…
7.11 %
covariancePredictor(Eigen::Matrix<float, 6, 6, 0, 6, 6>*, Eigen::Matrix<float, 6, 6, 0, 6, 6>*, …
33.95 %
kalmangainCalculator(Eigen::Matrix<float, 6, 6, 0, 6, 6>*, Eigen::Matrix<float, 2, 6, 0, 2, 6>*, Ei…
16.05 %
covarianceUpdate(Eigen::Matrix<float
, 2>*, Eigen::Matrix<float, 2, 6, 0, 2, 6
28.79 %
2 x
1 x
1 x
1 x
1 x

(The top most part of the graph is shown)

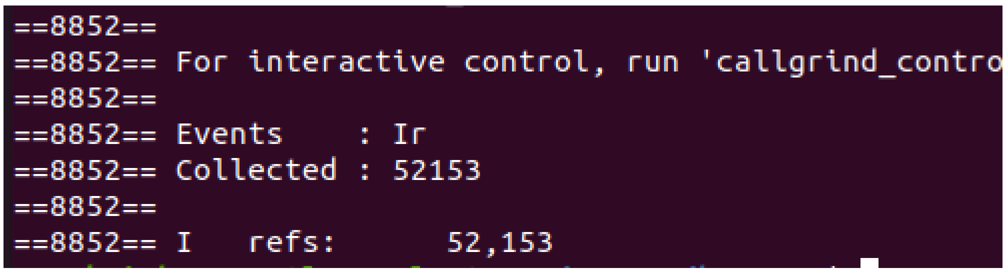
We observe that the covariancePredictor() and covarianceUpdate() functions take up the most of the time (around 33% and 28%). This is because of multiple matrix multiplication operations happening. Hence any optimization involved in the above two functions would have maximum impact.

If multi-threaded we would prefer both the functions to take place in parallel. But this may not be possible because the functions have to be executed sequentially.

We also observe that the measurementResidual() and stateUpdate() functions are not present in the graph. This is because both the functions involve negligible amount of time in comparison to the other functions.

Further analysis of each individual equation doesn’t seem possible because of the complexity involved with individual Eigen matrix operation. The call-graph is filled with Eigen library functions – which is difficult to analyze.

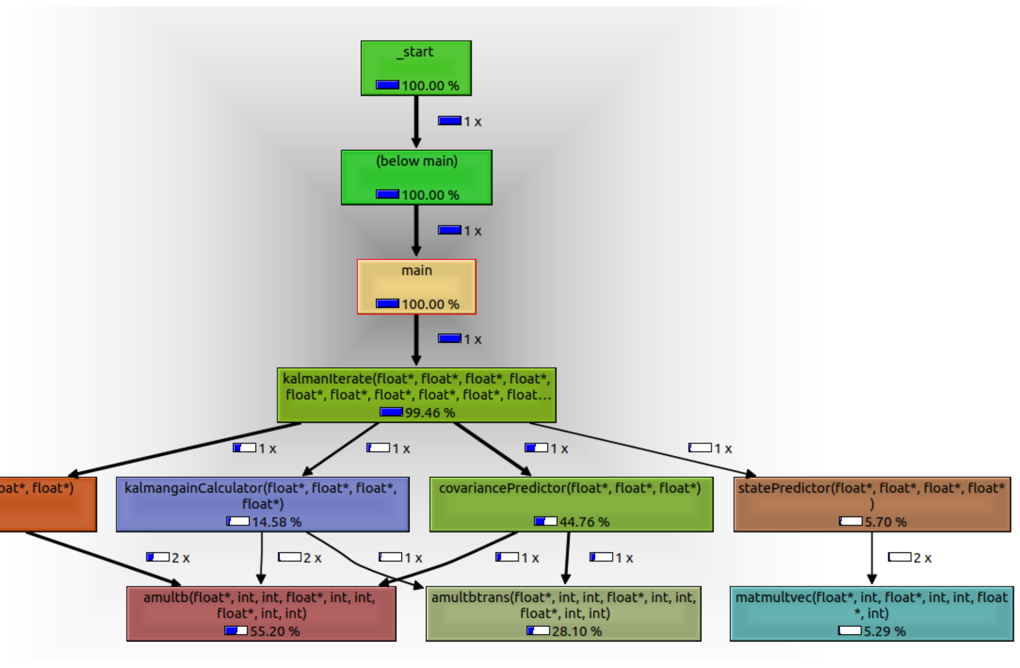
For further analysis of individual matrix operations in the six function equations, I profile the C code written by us (matrix operations were implemented by us) using the Valgrind and Callgrind tool. Following Call-graph was observed.


```
==8852==
==8852== For interactive control, run 'callgrind_contro
==8852==
==8852== Events    : Ir
==8852== Collected : 52153
==8852==
==8852== I   refs:      52,153
```

We observe that for smaller matrices, the code written by us is much faster and takes less instructions around 50k.

Note: the number of instructions might be lower when using Eigen library for matrix operations involving matrices of very large dimensions due to various optimizations.

From the Call-graph (**File**),


_start
100.00 %
1 x
(below main)
100.00 %
1 x
main
100.00 %
1 x
kalmanIterate(float*, float*, float*, float*, float*, float*, float*, float*, float*, float...
99.46 %
1 x
1 x
1 x
1 x
covarianceUpdate(float*, float*, float*)
31.78 %
kalmangainCalculator(float*, float*, float*, float*)
14.58 %
covariancePredictor(float*, float*, float*)
44.76 %
statePredictor(float*, float*, float*, float*)
5.70 %
2 x
2 x
1 x
1 x
1 x
2 x
amultb(float*, int, int, float*, int, int, float*, int, int)
55.20 %
amultbtrans(float*, int, int, float*, int, int, float*, int, int)
28.10 %
matmultvec(float*, int, float*, int, int, float *, int)
5.29 %

We can observe that again covariancePredictor() and covarianceUpdate() functions take up the most of the time. Within those functions – Matrix multiplications take up the most of the time.

Although the calculation of kalmangainCalculator() involves matrix inversion operations, the dimension of the matrix to be inverted is just 2*2 – hence doesn't shows up in the graph.

Note: We observe from the above Call-graphs that certain functions take more time than others. However it is important to note that – the call-graphs obtained are for a specific type of input: 6 dimensional state space, 2 dimensional input space. If the dimensions or the inputs were to change, the profiling analysis will definitely change.

Valgrind can also be used for analysis of multi-threaded code systems. Since OpenMP optimizations were already implemented in "Eigen" library – we profiled using Valgrind for our test case. Everything ran on a single thread – indicating for smaller matrices, Eigen doesn't do OpenMP optimizations.

Google-Performance Tools

We tried to do the same profiling analysis using gperf tools. But unfortunately we were never able to make it work. We tried with higher sampling frequencies as well, but no results showed up.

Conclusion

Based on the profiling analysis using Valgrind, we find functions to optimize and schedule appropriately in the DAG.