

NC State University
Department of Electrical and Computer Engineering
ECE 785: Spring 2019 (Dr. Dean)
Project #1:
Optimizing the Spherical Geometry Code
by
<< SALONI SHAMBHUWANI >>

NCSU Honor Pledge: "I have neither given nor received unauthorized aid on this test or assignment."

Student's electronic signature: __Saloni Shambhuwani____
(sign by typing your name)

Course number: _____ECE-785_____

Introduction

In this project I tried to optimize the execution speed of a program which find the closeness of capital from each waypoint. The base code gave nearly 74.821 us 73.250 us (average and minimum resp.) which is optimized to nearly 45 us and 43us. The function which were touched and optimized are: Calc_Closeness, sincos.c, Calc_Bearing.

Execution Time Optimization

Below is the table of all the optimizations performed to get the most optimized code.

Step	Otimization description	Time taken for optimization
1	Starter code	
2	Optimization1: Reducing Pipeline stall	5 hrs
3	Optimization2: Optimizing on cos function(cos approximation)	0.50 hr
4	Optimization3:Filteration of close points and optimization on fmod in cos function.	7-9hrs

#1: Starter code

The starter code gave 74.821 us 73.250 us as average and minimum time. So to start the optimization I looked into the perf record and found that the pipeline stalls to compare the max_c value and waiting for the comparison value to get ready. The stall can be seen in the image below:

```

P:\deban\dehbone\Project1\Project1.ag
Find_Nearest_Waypoint /home/debian/Project1/Project1.ag
0.00      ypop      (ds=did)
1.00      lmasa    eq, r4, r5, r6, r7, r8, r9, a1, r9, pc1
0.46      Calc_Clooseness(1)
0.46      close(p3-close - pi->lon);
0.24      vldm    r4, [r4, #12]
0.01      addw    r4, #40 ; dx20
0.01      vmla.f  r0, r0, a17
- bl      _cosf
0.03      pi->CosLat = p2->CosLat*
0.03      vldm    a14, [a4, #32] ; dfffffffo
0.73      return pi->SinLat * p2->SinLat +
0.73      vmla    r0, r0, #0 ; dfffffffo
0.03      pi->CosLat = pi->CosLat*
2.00      vmla.f  a14, a18, a14
0.00      return pi->SinLat * p2->SinLat +
0.73      vmla.f  a15, a19, a15
2.00      vmla.f  a15, a14, #0
Find_Nearest_Waypoint(1)
0.00      closest_i = 1;
0.73      vmla    a16, a15
0.46      vmla    a16, a16, fmasa
0.46      st      a16, r6
0.34      vmla    a16, a15
0.33      ldr
0.33      addw    r4, #1
0.33      b.n     lmasa_Find_Nearest_Waypoint=>dx4;
0.33      word    dx30=>dx4
0.33      word    dx45c7c1800
0.33      word    dx45b3ee0
0.33      word    dx4b40000
0.33      word    dx000183c
0.33      word    dx0000044

```

Here the pipeline stall can be seen to have spend about 89% of execution time.

debian@beaglebone: ~/Project/Project

Samples: 160K of event 'cycles:ppp', Event count (approx.): 3928242593

Overhead	Command	Shared Object	Symbol
62.90%	sg		[.] Find_Nearest_Waypoint
14.28%	sg		[.] cosf
0.54%	sg		[.] __kernel_cof
3.85%	[kernel.kallsyms]	[k]	_raw_spin_unlock_irqrestore
2.89%	sg		[.] ieee754_rem_pio2f
2.09%	sg		[.] __kernel_sinf
0.64%	[kernel.kallsyms]	[k]	sys_clock_gettime
0.58%	sg		[.] __acosf
0.54%	[kernel.kallsyms]	[k]	Vector_swi
0.48%	sg		[k] __do_div64
0.43%	sg		[.] __ieee754_sqrtf
0.39%	sg		[.] libc_d0_syscall
0.26%	sg		[.] __ieee754_atan2f
0.25%	sg		[.] __atanf
0.25%	sg		[.] __ieee754_acosf
0.16%	[kernel.kallsyms]	[k]	sample_to_timespec
0.15%	[kernel.kallsyms]	[k]	softirqentry_text_start
0.14%	sg		[k] arm_copy_to_user
0.13%	[kernel.kallsyms]	[k]	posix_cpu_clock_get_task
0.11%	[kernel.kallsyms]	[k]	ns_to_timespec_part1
0.10%	sg		[.] __clock_gettime
0.10%	sg		[.] main
0.09%	[kernel.kallsyms]	[k]	cpu_clock_sample
0.09%	[kernel.kallsyms]	[k]	ns_to_timespec
0.09%	[kernel.kallsyms]	[k]	task_sched_runtime
0.08%	sg		[k] preempt_count_sub
0.08%	[kernel.kallsyms]	[k]	thread_cpu_clock_get
0.06%	[kernel.kallsyms]	[k]	div_u64_rem
0.06%	[kernel.kallsyms]	[k]	_raw_spin_unlock_irq
0.06%	[kernel.kallsyms]	[k]	posix_cpu_clock_get
0.05%	[kernel.kallsyms]	[k]	ret_fast_syscall
0.05%	[kernel.kallsyms]	[k]	_lock_text_start
0.05%	[kernel.kallsyms]	[k]	task_rq_lock
0.02%	[kernel.kallsyms]	[k]	rcu_process_callbacks
0.01%	[kernel.kallsyms]	[k]	local_restart
0.01%	[kernel.kallsyms]	[k]	_udivsi3
0.01%	[kernel.kallsyms]	[k]	kmem_cache_free
0.01%	[kernel.kallsyms]	[k]	_und_usr
0.01%	[kernel.kallsyms]	[k]	Call_Fpe
0.01%	[kernel.kallsyms]	[k]	run_timer_softirq
0.01%	[kernel.kallsyms]	[k]	preempt_schedule_irq

Tip: Treat branches as callchains; perf report --branch-history

Also on looking at the report of performance we find that the program takes about 62.90% of time in function Find_Nearest_Waypoint(). So the optimization has to start from here. So below optimization is to reduce the pipeline stall and do some independent work to hide the memory latency.

- Also Highly accurate cosf function being used to calculate everything thus taking more time

VALIDATION

The validation code is made with double precision calculation of closeness and bearing and distance and is compared for any error in single precision code. The validation bit is set to zero in the code for now which is in geometry.h. but on making it 1 we can see the errors in distance and bearing for all capitals

Optimization #1: Reducing Pipeline stall

In this optimization, I tried to calculate the closeness from second waypoint for the capital while the program is calculating closeness for first waypoint(where the pipeline is stalling). On doing so I got the reduction in % time the program uses to Find_Nearest_Waypoint(). However the minium and average time remains the almost the same. This is because we are still calculating for all the waypoint it's just that we are doing two in one loop iteration. How much ever reduction is there is because I have removed one extra mov operation by using union of int and float value. Also using int reducing some of the floating point overhead and if you just

compare the two values by subtracting them then checking the sign of the value on the integer side it eliminates the stall. Since there is no way to get the FPSCR to the integer unit without stalling the pipeline, interacting with the FPSCR requires VMRS or VMSR instructions, which according to documents they stall the pipeline until all floating point instructions have completed. Thus the above optimization.

The perf report below shows the huge reduction from 89% for finding nearest point function. But cos looks to increase the overhead.

```

debian@beaglebone: ~/Project1/Project1
Samples: 255K of event 'cycles:ppp', Event count (approx.): 6250525657
Overhead Command Shared Object Symbol
-----
0.11% sg [.] cosl
14.33% sg [.] Find_Nearest_Waypoint
4.90% sg [.] do_cos.isra.0
2.43% sg [.] do_sin.isra.2
2.49% sg [kernel.kallsyms] [k] raw_spin_unlock_irqrestore
1.14% sg [.] _ieee754_atan2
1.12% sg [.] do_cos_slow.isra.1
0.75% sg [.] _ieee754_acos
0.71% sg [.] do_sin_slow.isra.3
0.65% sg [.] _sincosl
0.45% sg [kernel.kallsyms] [k] sys_clock_gettime
0.43% sg [kernel.kallsyms] [k] vector_swi
0.41% sg [.] _dubcos
0.31% sg [kernel.kallsyms] [k] _do_div64
0.24% sg [.] _libc_do_syscall
0.22% sg [.] _slow1
0.19% sg [.] _kernel_cosf
0.14% sg [kernel.kallsyms] [k] softirqentry_text_start
0.13% sg [kernel.kallsyms] [k] sample_to_timespec
0.11% sg [kernel.kallsyms] [k] posix_cpu_clock_get_task
0.10% sg [.] _doasin
0.10% sg [kernel.kallsyms] [k] asm_copy_to_user
0.09% sg [.] _kernel_sinf
0.08% sg [.] do_cos.isra.0
0.07% sg [.] _sincosf
0.07% sg [kernel.kallsyms] [k] _cleanup_count_sub
0.06% sg [.] main
0.06% sg [kernel.kallsyms] [k] ns_to_timespec.part.1
0.06% sg [kernel.kallsyms] [k] ns_to_timespec
0.06% sg [.] _dubsin
0.05% sg [kernel.kallsyms] [k] task_sched_runtime
0.05% sg [kernel.kallsyms] [k] raw_spin_unlock_irq
0.05% sg [kernel.kallsyms] [k] div_u64_rem
0.05% sg [.] _dubcos
0.05% sg [.] _clock_gettime
0.05% sg [kernel.kallsyms] [k] thread_cpu_clock_get
0.04% sg [kernel.kallsyms] [k] posix_cpu_clock_get
0.04% sg [kernel.kallsyms] [k] cpu_clock_sample
0.04% sg [kernel.kallsyms] [k] _lock_text_start
0.03% sg [kernel.kallsyms] [k] task_rq_lock
0.03% sg [.] _dubcos
Tip: For hierarchical output, try: perf report --hierarchy

```

From the perf record we can see that now the pipeline stalls in calculation of cosf() function. This is our next optimization then.

```

cosl /home/debian/Project1/Project1/sg
0.03 vmsr APSR_nxcr, fpscr
0.01 ldr r0, #0
0.00 vmlsl d7, d6, d5
0.00 vmlsl d7, d6, d5
0.00 vstr d7, [sp, #8]
0.00 vadd.f d7, d7, d0
0.01 vcmp.f d0, d7
0.00 vmsr APSR_nxcr, fpscr
0.15 beq.n 178f0 <__cos+0x176>
0.00 b.h 178f0 <__cos+0x164>
0.52 vabs.f d10, d8
0.17 vldr d0, [sp, #8]
0.35 vadd.f d7, d10, d0
0.16 vsub.f d0, d7, d0
0.17 vmov r5, r4
0.16 vsub.f d9, d10, d0
0.16 vmov.f d0, d9
0.14 -- bl do_cos.isra.0
4.73 vldr d7, [sp, #8]
36.34 vldr d6, [pc, #396] ; 17b78 <__cos+0x3ec>
0.30 vmla.f d5, d7, d6
0.15 vcmp.f d0, d5
0.15 vmsr APSR_nxcr, fpscr
0.15 beq.n 178f0 <__cos+0x176>
0.04 vldr d2, [pc, #260] ; 17b08 <__cos+0x37c>
0.01 mov r0, r5
0.01 add r1, sp, #16
0.00 vmov.f d0, d5
0.00 vmov.f d1, d2
0.00 -- bl do_cos_slow.isra.1
0.01 vldr d7, [sp, #16]
0.01 vadd.f d7, d0, d7
0.15 vcmp.f d0, d7
0.01 vmsr APSR_nxcr, fpscr
0.13 beq.n 178f0 <__cos+0x176>
0.00 vmov.f d0, d10
0.01 add r0, sp, #24
0.01 vldr d1, [pc, #216] ; 17b08 <__cos+0x37c>
0.01 -- bl docos
Press 'h' for help on key bindings

```

This is because the cos function is not optimized and floating point operations take a long time to perform as mentioned above.

Code change is shown below:

```
        c = Calc_Closeness(&ref, &(waypoints[i]) );
    (p.diff_float) = c - max_c;
    arr[i]=c;
    c1= Calc_Closeness(&ref, &(waypoints[i+1]) );
    (q.diff_float) = c1 - max_c;
    arr[i+1]=c1;
    // clock_gettime(CLOCK_THREAD_CPUTIME_ID, &end:
    if (!((p.diff_int) & 0x80000000)) {
        max_c = c;
        closest_i = i;
    }
    if (!((q.diff_int) & 0x80000000)) {
        //max_c = c1;
        closest_i = i+1;
    }
    if(max_c<c1)
    {
        max_c=c1;
    }
}
```

Optimization #2: Optimizing on cos function(cos approximation)

For cos approximations, I tried cos_32, cos_52, cos_73, cos_121 from sincos.c file and found that **cos_73 gives the lowest accuracy (and therefore fastest) version which provides correct results in validation**. The data for all 3 is attached in .txt files. It gave quite a good reduction in time and as shown below:

Now we see that the Find_Nearest_Waypoint() still take about 39% of program execution time So the next optimization goes for filtering the waypoints to closest with higher precision and finding these closest points as fast as possible .

```

debian@beaglebone: ~/Desktop/perfReport
cos_73 /home/debian/Desktop/perfReport/sg
// See the notes for an explanation of the range reduction.
//
double cos_73(double x)
{
1.08      push    {lr}
          -- bl    __gnu_mcount_nc
          int quad;
          //x=fmod(x, twopi);
          if(x<0)x=-x;
          quad=(int)(x * two_over_pi);
0.88      vldr    d16, [pc, #192] ; 10e58 <cos_73+0xc8>
          vabs.f d17, d0
          vmul.f d16, d17, d16
          vcvt.s s15, d16
0.71      vmov    s3, s15
          switch (quad)
03.96     cmp     r3, #3
          bhi.ln 10e52 <cos_73+0xc2>
          tbb     [pc, r3]
          .short 0x343b
          .short 0x021b
          {
          case 0: return  cos_73s(x);
          case 1: return -cos_73s(x-DF_PI-x);
          case 2: return -cos_73s(x-DF_PI);
          case 3: return  cos_73s(twopi-x);
0.06      vldr    d16, [pc, #160] ; 10e60 <cos_73+0xd0>
          cos_73s():
          return (c1 + x2*(c2 + x2*(c3 + x2*(c4 + c5*x2)))));
          vldr    d20, [pc, #172] ; 10e68 <cos_73+0xd8>
          cos_73():
          case 3: return  cos_73s(twopi-x);
Press 'h' for help on key bindings

```

```

debian@beaglebone: ~/Desktop/perfReport
Samples: 125K of event 'cycles:ppp', Event count (approx.): 30682806855
Overhead Command Shared Object Symbol
39.88% sg sg [.] cos_73
39.15% sg sg [.] Find_Nearest_Waypoint
4.76% sg sg [.] __gnu_mcount_nc
4.30% sg [kernel.kallsyms] [k] raw_spin_unlock_irqrestore
4.28% sg sg [.] __mcount_internal
0.83% sg [kernel.kallsyms] [k] sys_clock_gettime
0.65% sg [kernel.kallsyms] [k] vector_sw1
0.62% sg sg [.] sin_73
0.55% sg [kernel.kallsyms] [k] __do_div64
0.53% sg sg [.] ieee754_sqrtf
0.49% sg sg [.] ieee754_atan2f
0.47% sg sg [.] libc_do_syscall
0.33% sg sg [.] kernel_cosf
0.32% sg sg [.] atanf
0.26% sg sg [.] ieee754_acosf
0.20% sg [kernel.kallsyms] [k] sample_to_timespec
0.19% sg [kernel.kallsyms] [k] posix_cpu_clock_get_task
0.16% sg [kernel.kallsyms] [k] arm_copy_to_user
0.16% sg sg [.] __clock_gettime
0.15% sg sg [.] __sincof
0.14% sg [kernel.kallsyms] [k] __pollfdentry_text_start
0.14% sg sg [.] kernel_sinf
0.12% sg sg [.] main
0.12% sg [kernel.kallsyms] [k] ns_to_timespec.part.1
0.11% sg [kernel.kallsyms] [k] ns_to_timespec
0.10% sg [kernel.kallsyms] [k] preempt_count_sub
0.09% sg [kernel.kallsyms] [k] cpu_clock_sample
0.09% sg [kernel.kallsyms] [k] thread_cpu_clock_get
0.09% sg [kernel.kallsyms] [k] task_sched_runtime
0.08% sg [kernel.kallsyms] [k] div_s64_rem
0.08% sg [kernel.kallsyms] [k] posix_cpu_clock_get
Tip: Limit to show entries above 5% only: perf report --percent-limit 5

```

Optimization #3: Filtration of close points and optimization on fmod in cos function.

1. Refine your code to use two passes to filter out distant points.

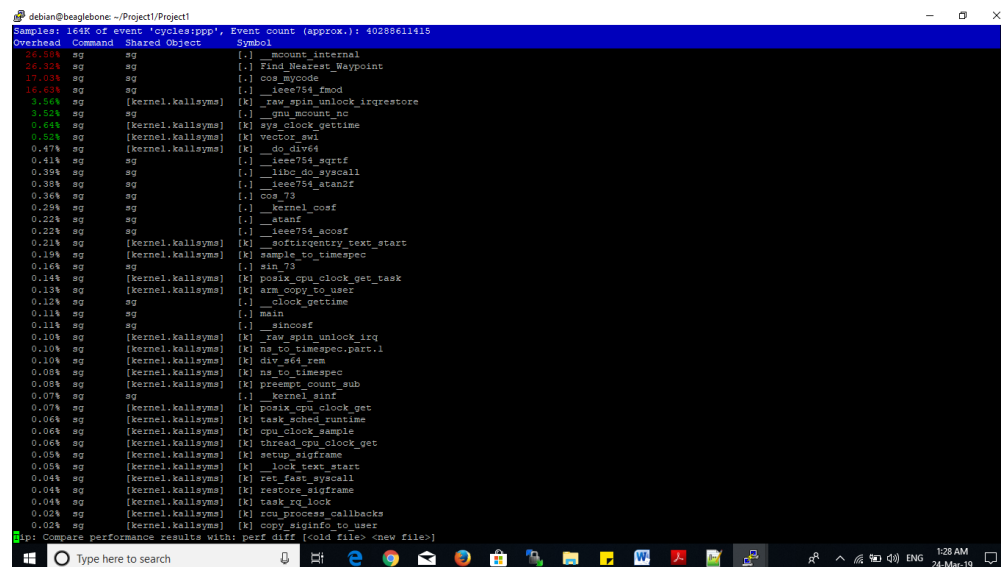
□ The first pass calculates the “closeness” to all the points using a fast but inaccurate cos approximation. Save the closeness of each point, and also the closeness of the closest point.

□ The second pass calculates closeness for all points with a closeness within ϵ of (apparent) closest point. The second pass calculates the “closeness” with a slower but more accurate cos approximation. You will need to calculate a valid error limit ϵ based on the maximum error of your approximation

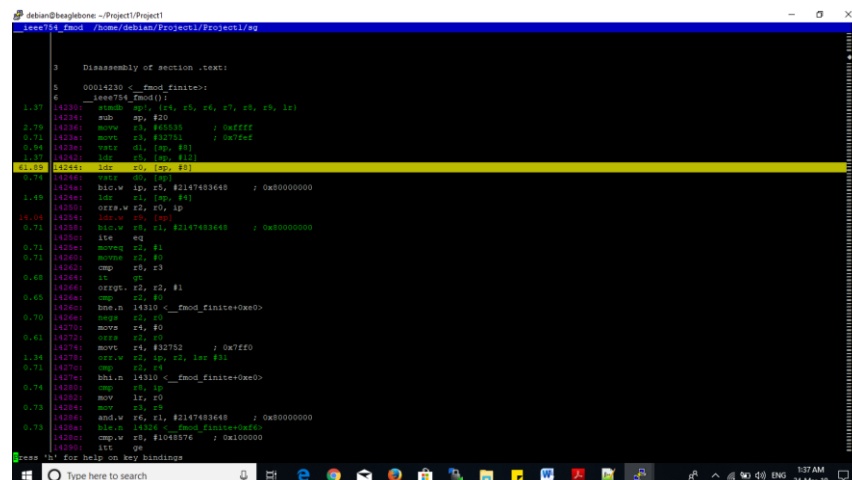
On first pass of the code I used cos_mycode which gives 1.2 decimal precision for cos to find the closest points. And then using cos_73 (precised and fast calculation version of cos) for

finding the closest point to capital point. Doing this reduced the overhead of Find_Nearest_Waypoint()

On looking as the perf annotate now I found that the overhead on mcount function is highest. To overcome this I removed the `-pg` from the Makefile which reduces the compiler overhead of profiling for gperf. This gave the reduction in time as shown below:



Below figure shows that the program time is most spent in calculation of fmod. Now still the overhead on cos was found to more. That means cos optimization still needs to be done. On looking at the object code of cos_73(), fmod is not required since the values are within 0 to $\pi/2$. So on reducing fmod overhead we get the time reduced to



```

Sample: 136K of event 'cycles:ppp', Event count (approx.): 3339821035
Overhead Command Shared Object Symbol
0.104 sg sg [.] _ieee754_fmod
0.074 sg sg [.] Find_Nearest_Waypoint
0.104 sg sg [.] cos_mycode
0.014 sg [kernel.kallsyms] [k] raw_spin_unlock_irqrestore
0.768 sg [kernel.kallsyms] [k] sys_clock_gettime
0.408 sg [kernel.kallsyms] [k] vector_sw1
0.008 sg [kernel.kallsyms] [k] do_div64
0.494 sg sg [.] _ieee754_sqrtf
0.444 sg sg [.] _ieee754_atan2f
0.444 sg sg [.] libc_60_syscall
0.374 sg sg [.] cos_73
0.384 sg sg [.] kernel_conf
0.284 sg sg [.] atanf
0.284 sg sg [.] _ieee754_acosf
0.224 sg [kernel.kallsyms] [k] sample_to_timespec
0.164 sg [kernel.kallsyms] [k] _softirqentry_test_start
0.164 sg [kernel.kallsyms] [k] posix_cpu_clock_get_task
0.154 sg [kernel.kallsyms] [k] arm_copy_to_user
0.154 sg sg [.] sin_73
0.114 sg sg [.] _sincof
0.124 sg [kernel.kallsyms] [k] ns_to_timespec.part.1
0.104 sg sg [.] _kernel_ninf
0.104 sg [kernel.kallsyms] [k] ns_to_timespec
0.104 sg sg [.] main
0.094 sg [kernel.kallsyms] [k] posix_cpu_clock_get
0.094 sg [kernel.kallsyms] [k] preempt_count_sub
0.094 sg sg [.] _clock_gettime
0.084 sg [kernel.kallsyms] [k] Thread_cpu_clock_get
0.084 sg [kernel.kallsyms] [k] cpu_clock_sample
0.074 sg [kernel.kallsyms] [k] task_sched_runtime
0.074 sg [kernel.kallsyms] [k] div_s64_rem
0.064 sg [kernel.kallsyms] [k] ret_fast_syscall
0.064 sg [kernel.kallsyms] [k] lock_text_start
0.064 sg [kernel.kallsyms] [k] task_rq_lock
0.054 sg [kernel.kallsyms] [k] raw_spin_unlock_irq
0.034 sg [kernel.kallsyms] [k] rcu_process_callbacks
0.024 sg [kernel.kallsyms] [k] _udivsi3
0.024 sg [kernel.kallsyms] [k] local_restart
0.014 sg [kernel.kallsyms] [k] kmem_cache_free
0.014 sg [kernel.kallsyms] [k] run_timer_softirq
0.014 sg [kernel.kallsyms] [k] queue_work_on
0.014 sg sg [.] _ieee754_rem_pio2f
0.014 sg [kernel.kallsyms] [k] _und_usr

```

The approximation table is shown below:

Function	Degree	Approximation	Epsilon (range)	# of points in pass 2
<i>cos_mycode</i>	2	$c_1 + c_2x^2$	0.011 – 0.6	7-8
<i>cos_32</i>	4	$c_1 + c_2x^2 + c_3x^4$	0 – 0.011	7-8
<i>cos_52</i>	6	$c_1 + c_2x^2 + c_3x^4 + c_4x^6$	0 – 0.003	5-6
<i>cos_73</i>	8	$c_1 + c_2x^2 + c_3x^4 + c_4x^6 + c_5x^8$	0 – 0.001	4-5

The final perf annotate and report can be seen below:

```

Sample: 103K of event 'cycles:ppp', Event count (approx.): 25344188032
Overhead Command Shared Object Symbol
0.754 sg sg [.] cos_mycode
0.474 sg sg [.] Find_Nearest_Waypoint
0.114 sg [kernel.kallsyms] [k] raw_spin_unlock_irqrestore
0.104 sg [kernel.kallsyms] [k] sys_clock_gettime
0.834 sg [kernel.kallsyms] [k] vector_sw1
0.014 sg sg [.] _sincof
0.404 sg sg [.] _kernel_conf
0.694 sg [kernel.kallsyms] [k] do_div64
0.604 sg sg [.] _ieee754_sqrtf
0.594 sg sg [.] libc_60_syscall
0.404 sg sg [.] _ieee754_atan2f
0.394 sg sg [.] cos_73
0.334 sg sg [.] atanf
0.224 sg sg [.] _ieee754_acosf
0.254 sg [kernel.kallsyms] [k] sample_to_timespec
0.214 sg sg [.] _kernel_ninf
0.214 sg [kernel.kallsyms] [k] posix_cpu_clock_get_task
0.194 sg [kernel.kallsyms] [k] arm_copy_to_user
0.174 sg sg [.] main
0.164 sg [kernel.kallsyms] [k] task_sched_runtime
0.154 sg [kernel.kallsyms] [k] _softirqentry_test_start
0.144 sg [kernel.kallsyms] [k] preempt_count_sub
0.134 sg [kernel.kallsyms] [k] ns_to_timespec.part.1
0.124 sg [kernel.kallsyms] [k] thread_cpu_clock_get
0.124 sg [kernel.kallsyms] [k] ns_to_timespec
0.104 sg [kernel.kallsyms] [k] div_s64_rem
0.104 sg sg [.] _clock_gettime
0.094 sg [kernel.kallsyms] [k] posix_cpu_clock_get
0.094 sg [kernel.kallsyms] [k] cpu_clock_sample
0.074 sg [kernel.kallsyms] [k] task_rq_lock
0.074 sg [kernel.kallsyms] [k] lock_text_start
0.064 sg [kernel.kallsyms] [k] ret_fast_syscall
0.064 sg [kernel.kallsyms] [k] raw_spin_unlock_irq
0.024 sg [kernel.kallsyms] [k] _udivsi3
0.024 sg [kernel.kallsyms] [k] local_restart
0.024 sg [kernel.kallsyms] [k] rcu_process_callbacks
0.014 sg [kernel.kallsyms] [k] kmem_cache_free
0.014 sg [kernel.kallsyms] [k] run_timer_softirq
0.014 sg [kernel.kallsyms] [k] queue_work_on
0.014 sg sg [.] _ieee754_rem_pio2f
0.014 sg [kernel.kallsyms] [k] _und_usr

```



```
debian@beaglebone: ~/Project1/Project1
cos_mycode /home/debian/Project1/Project1/sg

000108e0 <cos_mycode>:
cos_mycode():
float x2;                                // The input argument squared

x2=x * x;
return (c1 + x2*(c2 + c3 * x2));
}*/
0.45 float cos_mycode(float x){
    vabs.f s15, x0
    int quad;                            // what quadrant are we in?

    //x=fmod(x, twopi);                  // Get rid of values > 2* pi
    if(x<0)x=-x;                         // cos(-x) = cos(x)
    quad=(int) (x * two_over_pi);        // Get quadrant # (0 to 3) we're in

0.77 vldr d17, [pc, #136] ; 10970 <cos_mycode+0x90>
2.21 vcvrt.f d16, s15
0.80 vmsl.f d16, d16, d17
0.80 vcvrt.s s14, d16
0.80 vmov s3, s14
90.20 switch (quad){
0.75 cmp s3, #3
    bhl.n 10964 <cos_mycode+0x84>
    tbb [pc, s3]
    .word 0x020f1c29
    case 0: return cos_mycode(x);
    case 1: return -cos_mycode(OP_PI-x);
    case 2: return -cos_mycode(x-OP_PI);
    case 3: return cos_mycode(twopi-x);
0.06 vldr s14, [pc, #112] ; 10978 <cos_mycode+0x98>
cos_mycode():
{
const float c1= 0.9940307;
const float c2=-0.4855807;
float x2;                                // The input argument squared

x2=x * x;
return (c1 + x2*(c2));
0.00 vldr s13, [pc, #112] ; 1097c <cos_mycode+0x9c>
cos_mycode():
    case 3: return cos_mycode(twopi-x);
    vsub.f s15, s14, s15
cos_mycode():
Press 'h' for help on key bindings
```

Lessons learned:

The independent work can be done while the pipeline is stalling as the issue queue still has space to take the instructions.

The cos and sin approximations are the most important while dealing with floating point operations.

Floating point operations should be avoided as much as possible.