NC State University
Department of Electrical and Computer Engineering
ECE 785: Spring 2019 (Dr. Dean)
Project #2:

Vectorizing 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__________

Initial code analysis

In project1 the spherical geometry code was optimized for speed with initial speed being 73-74us reducing it to 44-45us. Now to further optimize we use SIMD instructions to get parallelism on the operations that are performed in the loop. In project1 the optimized code became stagnant (on optimization) for the three loops. The loops which were major area of interest were finding closeness using fast cos approximation and removing the farthest points, second loop to find the indexes of the closest points and final loop to find the closest of all the points using cos73s and thus calculating distance and bearing. These loops still took a lot of time in finding the closeness and stalling the pipeline.

Below is the table of optimizations performed and time taken to perform them and improvement after optimization:

Step	Otimization description	Time taken for optimization	Time reduction/Speed increase compared to previous optimization code
1	Starter code		
2	Optimization1: **Vectorization of cos function (fast cos (cos_mycode)**	5 hrs	~50%
3	Optimization2(a): **Vectorizing the array and radius comparison**	0.50 hr	~20%
4	Optimization2(b): **Vectorizing max value calculation and finding indexes.**	7hrs	~50%
5	Optimization3: **Inlining cos function**	10 mins	~1%

Optimization 1: Vectorization of cos function (fast cos (cos_mycode) - initial runs to find closest waypoints)

The cos function (fast cos – c1 + x2*(c2))in the project1 final code is called 164 times to find the closeness with less precision. Vectorizing this function will reduce the loop iteration by x/4. Following steps were performed to vectorize cos function:

- **i)** Inline the Calc_Closeness function into Find_Nearest_Waypoint so we can operate on four points at once.
- **ii)** Changed while loop into for loop iteration of 164, incrementing it loop induction by 4
- **iii)** Using vector multiplication found the values in each quadrant.
- **iv)** Using vector compare and select, found the correct quadrant for each element and selected the corresponding value for that quadrant (cos values found in step iii)

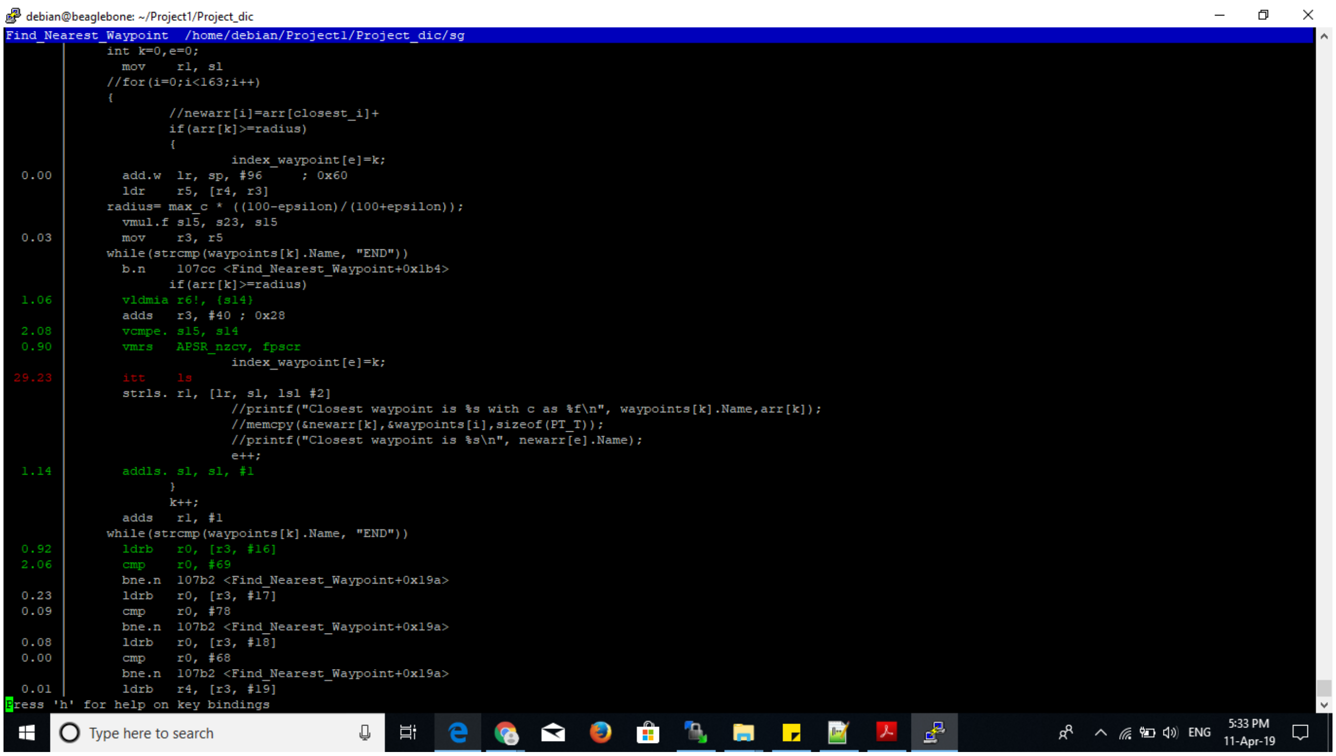
v) Unvectorized each value found from step iv and did the further operation for each lane separately.

vi) Include #include <arm_neon.h> in code and Changing the Makefile to make vectorized code.
MORE_CFLAGS = -fno-tree-loop-vectorize -fno-tree-vectorize -fno-unroll-loops

Thus on vectorizing, I found the significant reduction in time (almost by half). The final time after optimization is given below:

Time after optimization 1: 23.42-24.17us

Below is the Perf report after optimization:



```
debian@beaglebone: ~/Project1/Project_dic
Find_Nearest_Waypoint  /home/debian/Project1/Project_dic/sg
       │        int k=0,e=0;
       │          mov    r1, sl
       │        //for(i=0;i<163;i++)
       │        {
       │                //newarr[i]=arr[closest_i]+
       │                if(arr[k]>=radius)
       │                {
       │                        index_waypoint[e]=k;
  0.00 │          add.w  lr, sp, #96      ; 0x60
       │          ldr    r5, [r4, r3]
       │        radius= max_c * ((100-epsilon)/(100+epsilon));
       │          vmul.f s15, s23, s15
  0.03 │          mov    r3, r5
       │        while(strcmp(waypoints[k].Name, "END"))
       │          b.n    107cc <Find_Nearest_Waypoint+0x1b4>
       │                if(arr[k]>=radius)
  1.06 │          vldmia r6!, {s14}
       │          adds   r3, #40 ; 0x28
  2.08 │          vcmpe. s15, s14
  0.90 │          vmrs   APSR_nzcv, fpscr
       │                        index_waypoint[e]=k;
 29.23 │          itt    ls
       │          strls. r1, [lr, sl, lsl #2]
       │                        //printf("Closest waypoint is %s with c as %f\n", waypoints[k].Name,arr[k]);
       │                        //memcpy(&newarr[k],&waypoints[i],sizeof(PT_T));
       │                        //printf("Closest waypoint is %s\n", newarr[e].Name);
       │                        e++;
  1.14 │          addls. sl, sl, #1
       │                }
       │                k++;
       │          adds   r1, #1
       │        while(strcmp(waypoints[k].Name, "END"))
  0.92 │          ldrb   r0, [r3, #16]
  2.06 │          cmp    r0, #69
       │          bne.n  107b2 <Find_Nearest_Waypoint+0x19a>
  0.23 │          ldrb   r0, [r3, #17]
  0.09 │          cmp    r0, #78
       │          bne.n  107b2 <Find_Nearest_Waypoint+0x19a>
  0.08 │          ldrb   r0, [r3, #18]
  0.00 │          cmp    r0, #68
       │          bne.n  107b2 <Find_Nearest_Waypoint+0x19a>
  0.01 │          ldrb   r4, [r3, #19]
Press 'h' for help on key bindings
```


This shows that now the maximum time is spent in finding and storing the indexes in an array after fast cos. So we do the next optimization by vectorizing the array and radius comparison and finding index.

Optimization 2(a): Vectorizing the array and radius comparison

On vectorizing array and radius comparison there was few us reduction in time.Below steps were performed for vectorizing the comparison: Code is also shown below.

i) Loading first four array elements and creating radius as the vector of 4 elemnts.

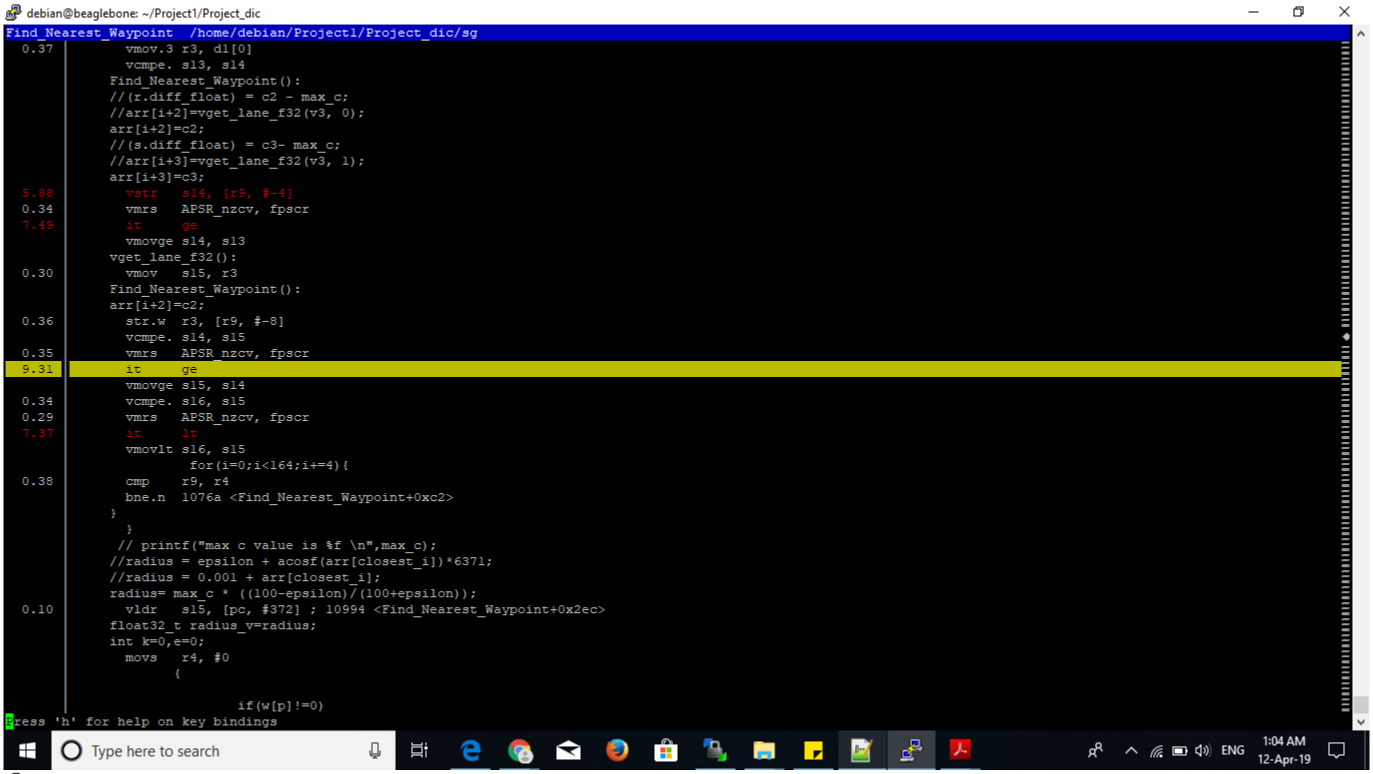
ii) Using vector compare instructions to compare 4 values in one iteration

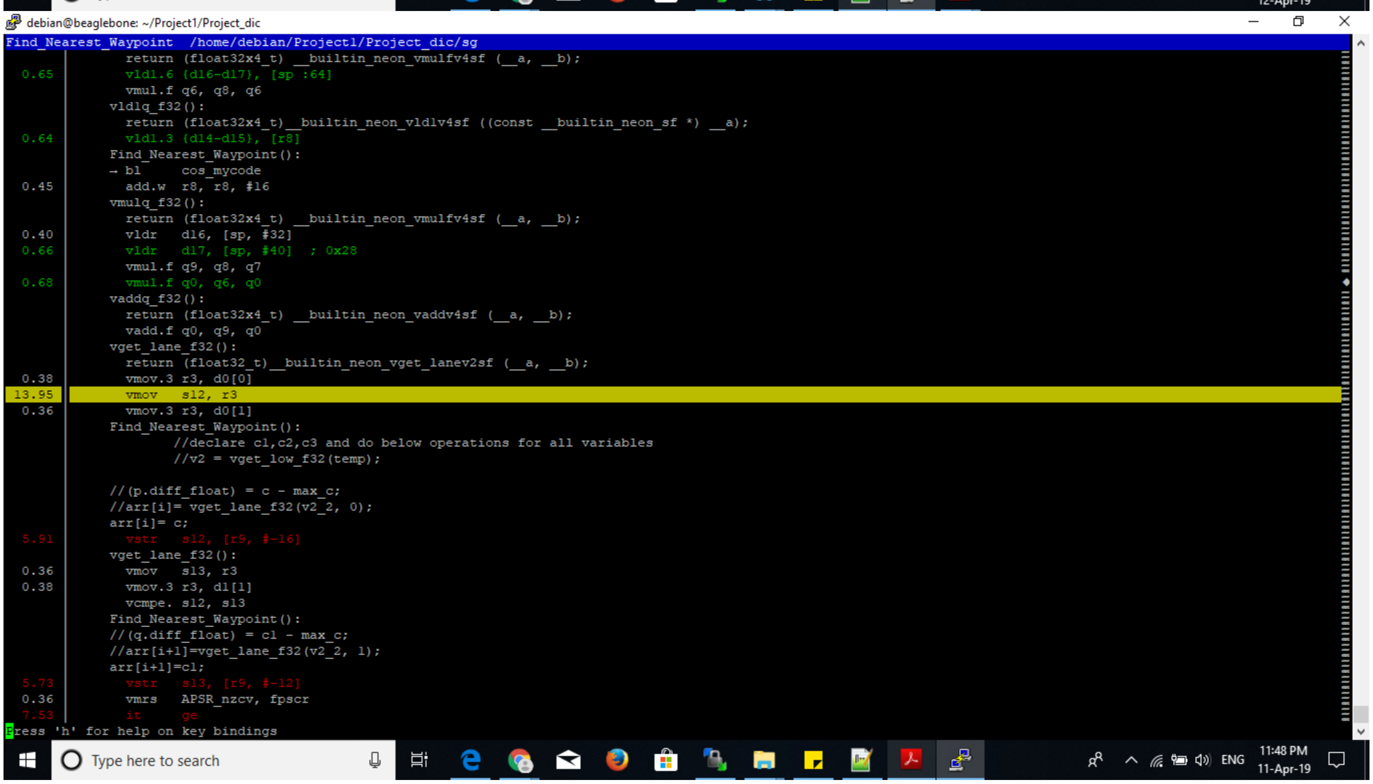
iii) Unvectorizing compare result and finding the indexes for the one's greater than radius.

```
for(k=0;k<164;k+=4)
{
    temp_arr=vld1q_f32(&(arr[k]));
    //print_float32x4(temp_arr);
    //printf("\n");
    temp_status=vcgeq_f32(temp_arr,temp_radius);
        //print_uint32x4(temp_status);
    //printf("\n");
    w1=vget_low_u32(temp_status);
    w[0]=vget_lane_u32(w1, 0);
    w[1]=vget_lane_u32(w1, 1);
    w1=vget_high_u32(temp_status);
    w[2]=vget_lane_u32(w1, 0);
    w[3]=vget_lane_u32(w1, 1);
    //printf("The values in w is %u %u %u %u",w[0],w[1],w[2],w[3]);
    for(int p=0;p<4;p++)
    {

        if(w[p]!=0)
        {

            index_waypoint[e]=k+p;
            e++;
        }
    }
```

This optimization/vectorization reduced the time by 6us. The time after optimization is 18.2 –18.7us.

Below perf report shows that the long time is spent in finding index and unvectorizing.





Optimization 2(b): Vectorizing max value calculation and finding indexes.

In this optimization I vectorized c_max value calculation and finding the indexes of the one's which are maximum. The optimization gave significant improvement. The time reduced to almost half. Below steps were performed in this optimization:

i) **Initializing the array vector with current indexes. Initializing previous c_max vector and index vector.**
ii) **Finding the c_max value using vector comparison between previous and current 4 lement valus of closeness.**
iii) **Masking the values with one if the previous and current value is same, else updating the values with current values.**
iv) **Updating the index vector for the values updated.**
v) **After loop completion forwarding c_max vector and index vector for 2nd pass.**

```
//vectorize c_max value calculation
//update the indexes for c_max value (selecting based on previous and new value)
//update if value has changed else take previous value
int array[]={i+0,i+1,i+2,i+3};
int32x4_t array_vector=vld1q_s32(&array[0]);
c_max_vector=vmaxq_f32(temp,c_max_vector);
uint32x4_t ene=vceqq_f32(c_max_vector_pre,c_max_vector);
//get indexes
max_value_indexes=vbslq_s32(ene,max_value_indexes,array_vector);
c_max_vector_pre=c_max_vector;
```

The code gave the reduction in time by half. Final time – 11.2-11.5us

Perf report is shown below after optimization 2(b).

debian@beaglebone: ~/Project1/Project_dic

```
Find_Nearest_Waypoint  /home/debian/Project1/Project_dic/sg
               cos_73(p2->Lon - p1->Lon);
  0.15       vldr   s0, [r1, #12]
               p1->CosLat * p2->CosLat*
             vmul.f s18, s15, s18
               cos_73(p2->Lon - p1->Lon);
  0.10       vldr   s15, [sp, #72]  ; 0x48
             vsub.f s0, s0, s15
  0.07       vcvt.f d0, s0
           → bl     cos_73
             return p1->SinLat * p2->SinLat +
  0.11       vcvt.f d8, s16
               p1->CosLat * p2->CosLat*
             vcvt.f d9, s18
             return p1->SinLat * p2->SinLat +
  0.14       vmla.f d8, d9, d0
  0.03       vcvt.f s16, d8
           Find_Nearest_Waypoint():
                   closest_i=w[t];
  0.08       vcmpe. s16, s20
  0.10       vmrs   APSR_nzcv, fpscr
 19.13       ite    gt
             movgt  r3, r8
  0.15       movle  r3, r7
             it     gt
  0.10       vmovgt s20, s16
           while(t<4)
             cmp.w  r9, #4
                   closest_i=w[t];
  0.07       mov    r7, r3
           while(t<4)
             beq.n  10866 <Find_Nearest_Waypoint+0x1be>
  0.05       ldr.w  r8, [r5, r9, lsl #2]
             b.n    10800 <Find_Nearest_Waypoint+0x158>
             // Finish calcuations for the closest point



           d = acosf(max_c)*6371; // finish distance calcuation
  0.48       vmov.f s0, s20
             str    r3, [sp, #0]
Press 'h' for help on key bindings
```

Type here to search

12:15 PM
13-Apr-19

debian@beaglebone: ~/Project1/Project_dic

```
Find_Nearest_Waypoint  /home/debian/Project1/Project_dic/sg
  0.12       mla    r4, r4, r7, r6
  0.07       vldr   s14, [sp, #72]  ; 0x48
             mov    r1, fp
             mov    r0, sl
           Find_Nearest_Waypoint():
           d = acosf(max_c)*6371; // finish distance calcuation
             vldr   s16, [pc, #140] ; 1090c <Find_Nearest_Waypoint+0x264>
           Calc_Bearing():
                   term1 = sinf(p1->Lon - p2->Lon)*p2->CosLat;
  0.10       vldr   s15, [r4, #12]
           Find_Nearest_Waypoint():
           d = acosf(max_c)*6371; // finish distance calcuation
             vmul.f s16, s0, s16
  0.08       vsub.f s0, s14, s15
           → bl     __sincosf
             vldr   s1, [sp, #88]   ; 0x58
           Calc_Bearing():
                   term1 = sinf(p1->Lon - p2->Lon)*p2->CosLat;
  6.10       vldr   s15, [r4, #8]
               p1->SinLat*p2->CosLat*cosf(p1->Lon - p2->Lon);
  0.05       vldr   s13, [sp, #68]  ; 0x44
             vmul.f s1, s1, s15
             vldr   s0, [sp, #92]   ; 0x5c
             term2 = p1->CosLat*p2->SinLat -
             vldr   s14, [r4, #4]
               p1->SinLat*p2->CosLat*cosf(p1->Lon - p2->Lon);
             vmul.f s1, s1, s13
             angle = atan2f(term1, term2) * (180/PI);
  0.01       vmul.f s0, s0, s15
             vldr   s15, [sp, #64]  ; 0x40
             vnmls. s1, s15, s14
           → bl     __atan2f_finite
  0.47       vldr   s15, [pc, #80]  ; 10910 <Find_Nearest_Waypoint+0x268>
             if (angle < 0.0)
  0.08       ldr    r3, [sp, #0]
           Find_Nearest_Waypoint():
           b = Calc_Bearing(&ref, &(waypoints[closest_i]) );
           *distance = d;
             *bearing = 360-b;
             *name = (char * ) (waypoints[closest_i].Name);
             movs   r2, #40 ; 0x28
Press 'h' for help on key bindings
```

Type here to search

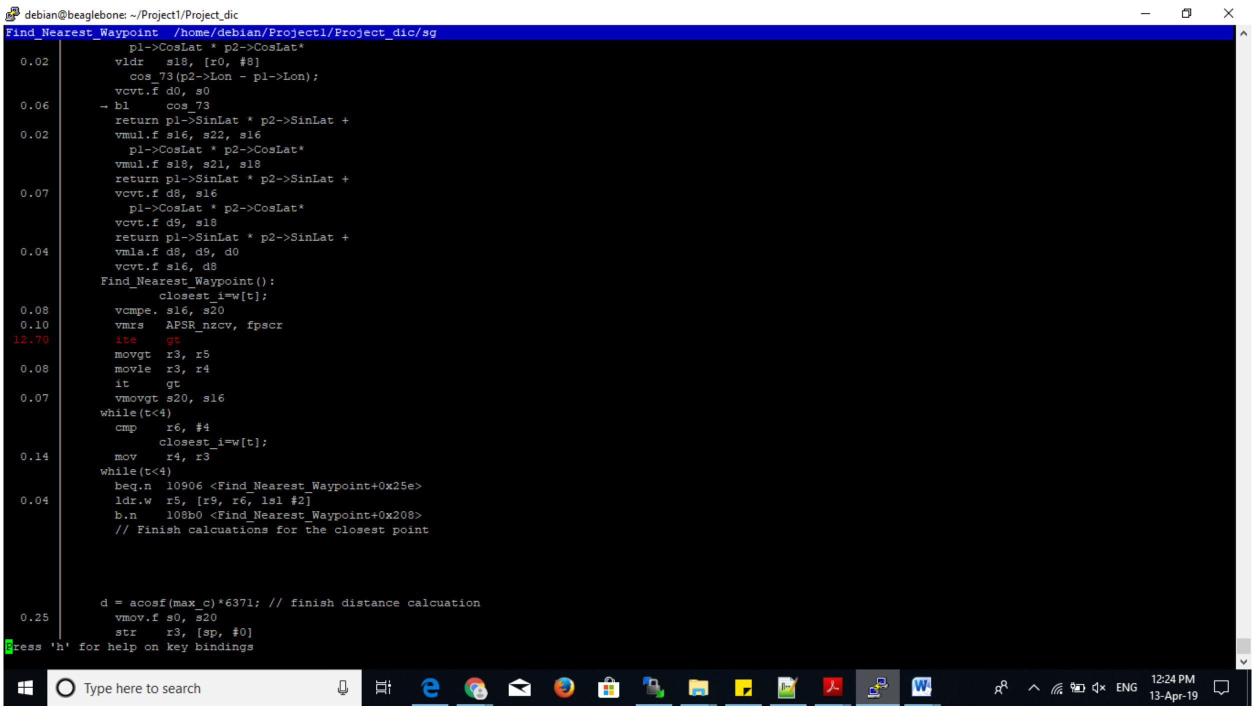
12:15 PM
13-Apr-19

Optimization 3: Inlining cos function

This optimization was more of the instinct than analysis.I was calling cos calculation function(vectorized cos-fast cos) from sincos.c. On inlining the cos function with geometry.c code it gave 1us reduction in time.

Final time: 10.08-10.3us

The perf report after optimization is shown below:



There is still time used up in the loop unvectorizing the indexes for 2nd pass. But second pass is double precision calculation. So the single precision vectorization could not be performed.

The vectorization combined with project1 optimization reduced the time from 75us to 10us which is the significant improvement in the speed of the code.

Lessons learned:

Optimisations get more and more difficult as time progresses as we need to find the inefficiencies at deeper and deeper level. Initially we can just look at the high-level code and figure out the optimizable sections, then we need to research a little bit more and get into the intricacies of the language. But, after that one needs to focus on the object code and see how the hardware/system behaves in presence of certain instructions. The programmer also needs to alter certain pre-made functions to better fit the application requirements thereby performing efficiently. In case of this code, it was already optimised to a very higher degree which made finding inefficiencies difficult.