Assignment 3

Programming for Performance, Group SELF

Naman Singla 200619 nsingla20@iitk.ac.in

Question 1

Consider the following code snippet.

```c
#define N (1 << 12)

double x[N], y[N], z[N], A[N][N];

for (int i = 0; i < N; i++) {
 for (int j = 0; j < N; j++) {
  y[j] = y[j] + A[i][j]*x[i];
  z[j] = z[j] + A[j][i]*x[i];
 }
}
```

Perform loop transformations to improve the performance of the following code snippet for sequential execution on one core (i.e., no multithreading). We will provide a template code to help with the computation.

1. Modify the `optimized()` function in the template code and perform evaluations.
2. Implement a version of the best-performing variant from the previous step using AVX2 intrinsics. Implement it as a separate function.

Briefly explain the reason for the poor performance of the provided reference version, your proposed optimizations (with justifications) to improve performance, and the improvements achieved. Time the different versions and report the speedup results with the GNU GCC compiler. Note that GCC does not auto-vectorize at optimization level `-O2`, you need to pass `-O3`. Evaluate part (i) with both optimization levels `-O2` and `-O3`.

For part (i), you may use any transformation we have discussed in class. For example, you can try to align statically or dynamically allocated memory and are free to use constructs like `__restrict__` and `#pragma GCC ivdep`. No array padding or layout transformation of arrays (e.g., 2D $\rightarrow$ 1D) are allowed.

Some transformations will lead to speedup, some will not. Include all the transformations that you tried, even if they did not work. Finally, summarize the set of transformations (e.g., 32 times unrolling + `xyz` permutation) that gave you the best performance.

Answer:

Reason for poor performance

Because the reference code does not take use of the data locality, it performs badly. Additionally, the matrix is accessed in column major order in the second statement of the innermost loop, which reduces the locality of reference.
We explore the spatial and temporal localization with the addition of loop fission and loop interchange. Increasing efficiency as a result.
Moreover, vectorization of code is also not possible because of column-major access and A being a double pointer. Loop fission and interchange will solve the column major access problem but to resolve double pointer we will force compiler to vectorize the loop using pragma ivdep and declare all arrays as global static alligned arrays. We will also add restrict attribute to variables to prevent aliasing.

1. **Optimized Version**

```
int i, j;
  for (i = 0; i < N; i++) {
    #pragma GCC ivdep
    for (j = 0; j < N; j++) {
      y_opt[j] = y_opt[j] + A[i][j] * x[i];

    }
  }
  for(j = 0; j < N; j++){
    #pragma GCC ivdep
    for(i=0;  i < N; i++) {
      z_opt[j] = z_opt[j] + A[j][i] * x[i];
    }
  }
```

Transformation : **Loop fission + Loop Interchange**

O2			**O3**		
Ref Time	OP time	Speedup	Ref Time	OP time	Speedup
0.74167	0.10521	7.05	0.35371	0.09814	3.604

NOTE:- As opposed to question I observed that loop was vectorized both in O2 and O3. May be due to ivdep.

2. **AVX Version**

```
for (i = 0; i < N; i++) {
    base = A[i];
    x_avx = _mm256_set1_pd(x[i]);
    for (j = 0; j < N; j += 4) {
      A_avx = _mm256_load_pd(base+j);
      y_avx = _mm256_load_pd(y_opt+j);
      y_avx = _mm256_fmadd_pd(A_avx, x_avx, y_avx);
      _mm256_storeu_pd(y_opt+j, y_avx);
    }
  }

  for (j = 0; j < N; j++) {
    base = A[j];
    x_avx = _mm256_set1_pd(x[j]);
    for (i = 0; i < N; i += 4) {
      A_avx = _mm256_load_pd(base+i);
      z_avx = _mm256_load_pd(z_opt+i);

      z_avx = _mm256_fmadd_pd(A_avx, x_avx, z_avx);
      _mm256_storeu_pd(z_opt+i, z_avx);
    }
  }
```

Transformation : **Loop fission + Loop Interchange**

O2			**O3**		
Ref Time	OP time	Speedup	Ref Time	OP time	Speedup
0.74167	0.06341	11.696	0.35371	0.06322	5.595

The same boost in performance in O2 and O3 as expected as we are using intrinsics

NOTE:- Although Ref Time is coming 0.7 or 0.3, it is due to optimization that compiler automatially applied. With no optimizations Ref time = 1.8 sec

Question 3

Problem 3 **[20+40+40 marks]**

Implement versions of an *inclusive* prefix sum function using (i) SSE4 intrinsics, (iii) AVX2 instrinsics, and (iv) AVX-512 intrinsics. Compare the performance with the sequential and OpenMP versions, and report performance speedups.

Include the compiler version and the exact compilation command. It is okay to tweak the provided compilation command to suit your compiler version and the target architecture. You should refer to the GCC manual for details.

Use the template code provided with the description. There are multiple algorithms to compute prefix sum in parallel. Try to come up with implementations that perform the best for you (e.g., allow the compiler to vectorize the loops).

Refer to `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#` for documentation on Intel Intrinsics.

Answer: NOTE :- to get visible difference I will use #define N (1 « 30). All units are in microseconds

1. **SEE4**
   The template code provided for this part is completely correct expect for the stride part. A multiple of 8 is missing for the calculation to convert bytes to bits. Provided code is very efficient, performs 2 shit operation and additions to calculate the prefix sum for 4 elements.

Serial	OMP	SEE
723652	696499	663863

2. **AVX2** Our aim would be to implement the most efficient shift operations, since avx2 doesn't support inter-lane shift operations.

   (a) V1
   AVX2 doesn't support inter lane shift operations but it does support permutation in 2 lanes.

```
 //4 bytes
 tmp = _mm256_permutevar8x32_epi32(x, _mm256_set_epi32(6, 5, 4,
3, 2, 1, 0, 0));
 tmp = _mm256_and_si256(tmp, _mm256_set_epi32(-1, -1, -1, -1, -1,
 -1, -1, 0));
 x = _mm256_add_epi32(x, tmp);

 //8 bytes
 tmp = _mm256_permutevar8x32_epi32(x, _mm256_set_epi32(5, 4, 3,
2, 1, 0, 0, 0));
 tmp = _mm256_and_si256(tmp, _mm256_set_epi32(-1, -1, -1, -1, -1,
 -1, 0, 0));
```

```
    x = _mm256_add_epi32(x, tmp);

    //16 bytes
    tmp = _mm256_permutevar8x32_epi32(x, _mm256_set_epi32(4, 3, 2,
   1, 0, 0, 0, 0));
    tmp = _mm256_and_si256(tmp, _mm256_set_epi32(-1, -1, -1, -1, 0,
   0, 0, 0));
    x = _mm256_add_epi32(x, tmp);

```

Permute is followed by and operation to get zero-filling effect of shift.

(b) V2
Going by same logic another way to implement shift operations is as follows:

```
    // Shift 4 bytes
    tmp = _mm256_permute2x128_si256(x, x, _MM_SHUFFLE(0,0,3,0) );
    tmp = _mm256_alignr_epi8(x,tmp,12);
    x = _mm256_add_epi32(x, tmp);

    //Shift 8 bytes
    tmp = _mm256_permute2x128_si256(x, x, _MM_SHUFFLE(0,0,3,0) );
    tmp = _mm256_alignr_epi8(x,tmp,8);
    x = _mm256_add_epi32(x, tmp);

    //Shift 16 bytes
    tmp = _mm256_permute2x128_si256(x, x, 41);
    x = _mm256_add_epi32(x, tmp);

```

(c) V3
Another way would be to process 2 128 bits lanes separately and then add the offset to higher position lane

```
    // parallel 128 bits
    x = _mm256_add_epi32(x, _mm256_bslli_epi128(x, 4));
    x = _mm256_add_epi32(x, _mm256_bslli_epi128(x, 8));

    __m256i tmp = _mm256_permutevar8x32_epi32(x, _mm256_set_epi32(3,
    3, 3, 3, 0, 0, 0, 0));
    tmp = tmp = _mm256_and_si256(tmp, _mm256_set_epi32(-1, -1, -1,
   -1, 0, 0, 0, 0));
    x = _mm256_add_epi32(x, tmp);

```

NOTE :- I am using permutevar8x32 because it is one of the few operations that apply to all 256 bits simultaneously(bot as 2 128 bit lanes).

The performance advantages of avx2 over see4 are limited by the fact that there is a dependence on previous iteration. V3 is a good choice as it tries to treat lanes differently. But to improve further the offset calculation can be done in m128i to reduce the cycles (avx takes 2 cycle, whereas we only need to operate on 128 bits).

```
    // parallel 128 bits
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 4));
    x = _mm256_add_epi32(x, _mm256_slli_si256(x, 8));

    __m128i l,h;
    l = _mm256_extractf128_si256(x,0);
    l = _mm_add_epi32(l, offset);
    offset = _mm_shuffle_epi32(l, mask);

    h = _mm256_extractf128_si256(x,1);
    h = _mm_add_epi32(h, offset);
    offset = _mm_shuffle_epi32(h, mask);

    x = _mm256_set_m128i(h, l);

```

Lets compare all versions

Serial	OMP	V1	V2	V3	AVX+SEE4
723652	696499	663975	665306	660073	655470

3. **AVX512**
   I will extend V3 of AVX2 because now the number of 128 lanes has increased, its better to handle in one go rather than operating still at 128 bit level. More, AVX512 provide a newer premute instruction that don't require us to take AND to zero the elements i.e masking.

```
__mmask16 s4 = _cvtu32_mask16(0b1111111111111110);
__m512i p4 = _mm512_set_epi32(14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
     1, 0, 0);

__mmask16 s8 = _cvtu32_mask16(0b1111111111111100);
__m512i p8 = _mm512_set_epi32(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
    0, 0, 0);

__mmask16 s16 = _cvtu32_mask16(0b1111111111110000);
__m512i p16 = _mm512_set_epi32(11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0,
    0, 0, 0);

__mmask16 s32 = _cvtu32_mask16(0b1111111100000000);
__m512i p32 = _mm512_set_epi32(8, 7, 6, 5, 4, 3, 2, 1, 0, 0, 0, 0, 0, 0,
    0, 0);

```

Above will be the masks we will use and remain const during entire operation.
Code :

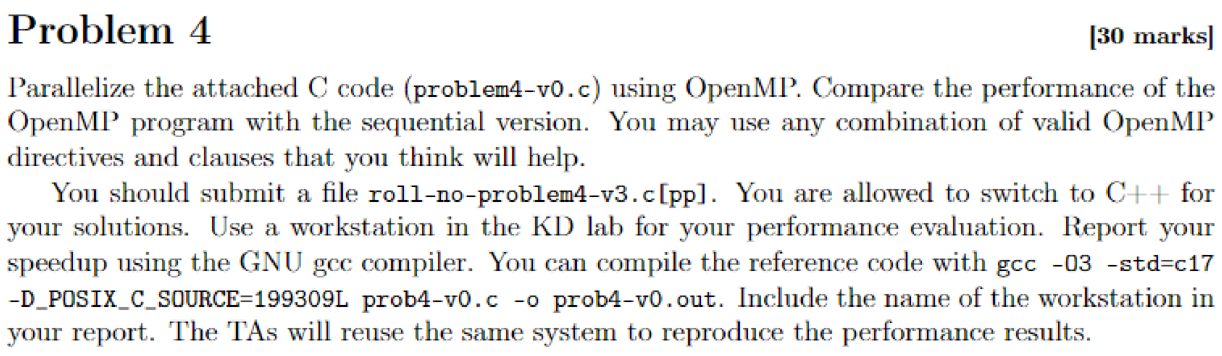
```
    // shift 4 bytes
    __m512i tmp = _mm512_maskz_permutexvar_epi32(s4, p4, x);
    x = _mm512_add_epi32(x, tmp);

    // shift 8 bytes
    tmp = _mm512_maskz_permutexvar_epi32(s8, p8, x);
    x = _mm512_add_epi32(x, tmp);

    // shift 16 bytes
    tmp = _mm512_maskz_permutexvar_epi32(s16, p16, x);
    x = _mm512_add_epi32(x, tmp);

    // shift 32 bytes
    tmp = _mm512_maskz_permutexvar_epi32(s32, p32, x);
    x = _mm512_add_epi32(x, tmp);

    x = _mm512_add_epi32(x, offset);

```

Serial	OMP	AVX512
709328	670915	599337

Complete comparison :

```
Serial version: 1073741824 time: 704249
OMP version: 1073741824 time: 671073
SSE version: 1073741824 time: 629319
AVX2 version: 1073741824 time: 621985
AVX2 version1: 1073741824 time: 627829
AVX2 version2: 1073741824 time: 635351
AVX2 version3: 1073741824 time: 622655
AVX512 version: 1073741824 time: 597104
```

Question 4



Problem 4 **[30 marks]**

Parallelize the attached C code (`problem4-v0.c`) using OpenMP. Compare the performance of the OpenMP program with the sequential version. You may use any combination of valid OpenMP directives and clauses that you think will help.

You should submit a file `roll-no-problem4-v3.c[pp]`. You are allowed to switch to C++ for your solutions. Use a workstation in the KD lab for your performance evaluation. Report your speedup using the GNU gcc compiler. You can compile the reference code with `gcc -O3 -std=c17 -D_POSIX_C_SOURCE=199309L prob4-v0.c -o prob4-v0.out`. Include the name of the workstation in your report. The TAs will reuse the same system to reproduce the performance results.

Answer:

- **Reason for poor performance**
  The main reason for poor performance is non-utilisation of cache model. In this code memory is being accesses are vary far from each other (no spatial locality). There are also some unnecessary variables which can be removed.

- **Specification of system used**

```
Static hostname: csews54
       Icon name: computer-desktop
         Chassis: desktop
      Machine ID: 456294d50a3447caa9bac60dbf33dd8f
         Boot ID: c1ae26eab99845bcbe0ac98accc50c4f
Operating System: Ubuntu 22.04.1 LTS
          Kernel: Linux 5.15.0-50-generic
    Architecture: x86-64
 Hardware Vendor: Lenovo
  Hardware Model: ThinkCentre M920t
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
    CPU family:          6
    Model:               158
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            10
    CPU max MHz:         4600.0000
    CPU min MHz:         800.0000
    BogoMIPS:            6399.96
```

```
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   192 KiB (6 instances)
  L1i:                   192 KiB (6 instances)
  L2:                    1.5 MiB (6 instances)
  L3:                    12 MiB (1 instance)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0-11
Vulnerabilities:
  Itlb multihit:         KVM: Mitigation: VMX disabled
  L1tf:                  Mitigation; PTE Inversion; VMX conditional cache
     flushes, SMT vulnerable
  Mds:                   Mitigation; Clear CPU buffers; SMT vulnerable
  Meltdown:              Mitigation; PTI
  Mmio stale data:       Mitigation; Clear CPU buffers; SMT vulnerable
  Retbleed:              Mitigation; IBRS
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled
    via prctl and seccomp
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user
    pointer sanitization
  Spectre v2:            Mitigation; IBRS, IBPB conditional, RSB filling,
     PBRSB-eIBRS Not affected
  Srbds:                 Mitigation; Microcode
  Tsx async abort:       Mitigation; TSX disabled
```

I will be directly using the optimized code (in Part ii) that we arrived at in Assignment 2(see Appenddix for approach). We will add the following line on outermost loop to parallelize the code.

```
#pragma omp parallel for default(none) shared(c,b,fptr,s,e) private(qb)
   reduction(+:pnts)

```

By doing so outer loop will get parallelized. I verified that there are 14 iteration in each for loop > 12(Number of cores in system). While running the code I observed that all CPUs were 100% initially but towards the end only 2 CPUs remain in use. Hence, we can say that it can be optimized further to some extent.

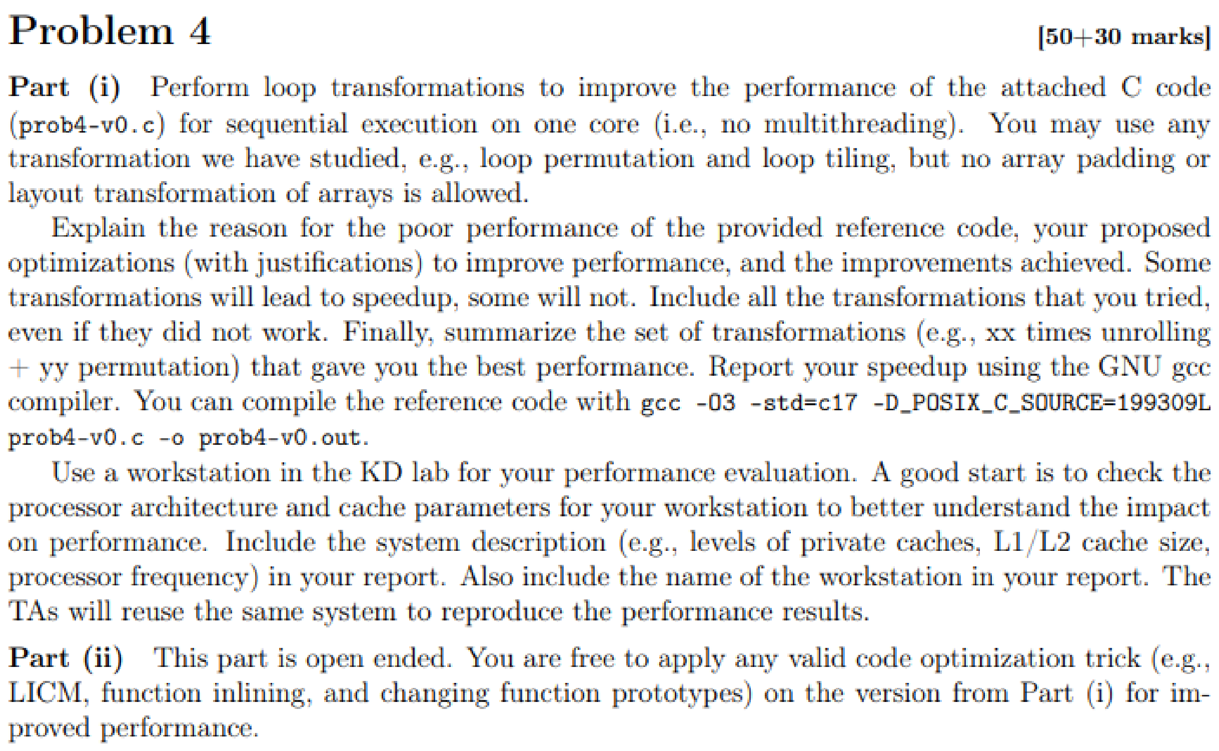
Ref Time	OP time(Seq)	OP time (parallel)	Speedup
417.07 seconds	169.98 seconds	36.32 seconds	11.58

There is nothing much left to optimize in single iteration expect that calculation of qb can be vectorized. I tried to vectorized it using "#pragma omp simd" but that resulted in reduced performance whereas force vectorization using "#pragma vector always" is not vectorizing(may be due to 2D array index and compiler is not able to prove dependencies).

Let's try to utilize the CPU completely. For this purpose, we will use nested parallelism. We will parallelize r1 and r5 so that all the cores remain buzy (200619-prob4-v3.2.c)
Total time = 32.734880 seconds, Speedup: 13x (Not much difference).

I also tried to collapse and parallelize. It will utilize cores more efficiently but we will have to repeat same calculations multiple times. That overall lead to increase in time taken (200619-prob4-v3.1.c).
Total time = 150.761507 seconds

Appendix



Problem 4 **[50+30 marks]**

Part (i) Perform loop transformations to improve the performance of the attached C code (`prob4-v0.c`) for sequential execution on one core (i.e., no multithreading). You may use any transformation we have studied, e.g., loop permutation and loop tiling, but no array padding or layout transformation of arrays is allowed.

Explain the reason for the poor performance of the provided reference code, your proposed optimizations (with justifications) to improve performance, and the improvements achieved. Some transformations will lead to speedup, some will not. Include all the transformations that you tried, even if they did not work. Finally, summarize the set of transformations (e.g., xx times unrolling + yy permutation) that gave you the best performance. Report your speedup using the GNU gcc compiler. You can compile the reference code with `gcc -O3 -std=c17 -D_POSIX_C_SOURCE=199309L prob4-v0.c -o prob4-v0.out`.

Use a workstation in the KD lab for your performance evaluation. A good start is to check the processor architecture and cache parameters for your workstation to better understand the impact on performance. Include the system description (e.g., levels of private caches, L1/L2 cache size, processor frequency) in your report. Also include the name of the workstation in your report. The TAs will reuse the same system to reproduce the performance results.

Part (ii) This part is open ended. You are free to apply any valid code optimization trick (e.g., LICM, function inlining, and changing function prototypes) on the version from Part (i) for improved performance.

Answer:
Part(i)

```
s1 = floor((dd2 - dd1) / dd3);
s2 = floor((dd5 - dd4) / dd6);
...
s10 = floor((dd29 - dd28) / dd30);

for (int r1 = 0; r1 < s1; ++r1) {
  x1 = dd1 + r1 * dd3;

  for (int r2 = 0; r2 < s2; ++r2) {
    x2 = dd4 + r2 * dd6;
        ...
                for (int r10 = 0; r10 < s10; ++r10) {
                  x10 = dd28 + r10 * dd30;

                  q1 = fabs(c11 * x1 + c12 * x2 + ... + c110 * x10 - d1);
```

Total time = 421.382426 seconds

(a) Making it a perfect loop nest won't help as the inner statement gets executed more times.

Lets try to minimize the the number of variables. Loops index variables r_n depend on s_n for upper limits and inner statements use only x_n that is calculated from r_n. We can note that s_n has same dd_n as multiplied by r_n to obtain x_n. So, we can completely remove s_n by incrementing r_n by dd_n instead of 1. Upper and lower limit can be same as used to calculate s_n.
After this we can also see that $x_n = r_n$ now. Hence, replace x with r. And what we got is a perfect nest with reduce number of variables.

```c
for (r1 = dd1; r1 < dd2; r1+=dd3) {
    for (r2 = dd4; r2 < dd5; r2+=dd6) {
            ....
                    for (r10 = dd28; r10 < dd29; r10+=dd30) {

                      q1 = fabs(c11 * r1 + c12 * r2 + ... + c110 * r10 -
    d1);
```

Total time = 361.630364 seconds, Speedup : 1.16x (200619-prob4-v1-1.c)
NOTE : I will take above as reference now

(b) Since there is not data dependence in inner-most loop, we can permute the loops in any manner. I tried reversing the order of r_n by time taken increased significantly

```c
for (r10 = dd28; r10 < dd29; r10+=dd30)
    for (r9 = dd25; r9 < dd26; r9+=dd27)
      for (r8 = dd22; r8 < dd23; r8+=dd24)
        for (r7 = dd19; r7 < dd20; r7+=dd21)
          for (r6 = dd16; r6 < dd17; r6+=dd18)
            for (r5 = dd13; r5 < dd14; r5+=dd15)
              for (r4 = dd10; r4 < dd11; r4+=dd12)
                for (r3 = dd7; r3 < dd8; r3+=dd9)
                  for (r2 = dd4; r2 < dd5; r2+=dd6)
                    for (r1 = dd1; r1 < dd2; r1+=dd3)

```

Total time = 556.030280 seconds (200619-prob4-v1-2.c)

(c) I tried unrolling innermost r10 loop but it doesn't show any improvement in performance. May be due to the jump inserted using conditional statement and my index variables are not integers.
Total time = 413.894837 seconds (200619-prob4-v1-unrol.c)

(d) We don't have arrays that's why most optimizations are not implementable. Index of loops are used as variables in calculations rather than being used as index of some array. We can take advantage of this by scalar expansion. We can store some precomputed calculation at each step. Continuing from (a) , We can store the value of q_n of parent loop and then do computation in outer loops.

```c
for (r1 = dd1; r1 < dd2; r1+=dd3) {
    qb[1][1] = c11 * r1 - d1;
    qb[1][2] = c21 * r1 - d2;
    ...
    qb[1][10] = c101 * r1 - d10;
    for (r2 = dd4; r2 < dd5; r2+=dd6) {
      qb[2][1] = qb[1][1] + c12 * r2;
      qb[2][2] = qb[1][2] + c22 * r2;
      ...
      qb[2][10] = qb[1][10] + c102 * r2;

        ....
                    for (r10 = dd28; r10 < dd29; r10+=dd30) {
                      q1 = fabs(qb[10][1]);
                      q2 = fabs(qb[10][2]);
                      ...
                      q10 = fabs(qb[10][10]);
```

Total time = 166.152974 seconds, Speedup : 2.5x (200619-prob4-v1.c)
10 Scalar expansion + LICM

(e) Apart from this I also tried to maximize the cache hits. But those programs had less or no impact on performance. (200619-prob4-v1-3.c) (200619-prob4-v1-3-1.c)

- **Reason for poor performance**
  The main reason for poor performance is non-utilisation of cache model. In this code memory is being accesses are vary far from each other (no spatial locality). There are also some unnecessary variables which can be removed.

- **Specification of system used**

```
Static hostname: csews54
       Icon name: computer-desktop
         Chassis: desktop
      Machine ID: 456294d50a3447caa9bac60dbf33dd8f
         Boot ID: c1ae26eab99845bcbe0ac98accc50c4f
Operating System: Ubuntu 22.04.1 LTS
          Kernel: Linux 5.15.0-50-generic
    Architecture: x86-64
 Hardware Vendor: Lenovo
  Hardware Model: ThinkCentre M920t
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
    CPU family:          6
    Model:               158
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            10
    CPU max MHz:         4600.0000
    CPU min MHz:         800.0000
    BogoMIPS:            6399.96
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   192 KiB (6 instances)
  L1i:                   192 KiB (6 instances)
  L2:                    1.5 MiB (6 instances)
  L3:                    12 MiB (1 instance)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0-11
Vulnerabilities:
  Itlb multihit:         KVM: Mitigation: VMX disabled
  L1tf:                  Mitigation; PTE Inversion; VMX conditional cache
    flushes, SMT vulnerable
  Mds:                   Mitigation; Clear CPU buffers; SMT vulnerable
  Meltdown:              Mitigation; PTI
  Mmio stale data:       Mitigation; Clear CPU buffers; SMT vulnerable
  Retbleed:              Mitigation; IBRS
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled
   via prctl and seccomp
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user
   pointer sanitization
  Spectre v2:            Mitigation; IBRS, IBPB conditional, RSB filling,
    PBRSB-eIBRS Not affected
  Srbds:                 Mitigation; Microcode
  Tsx async abort:       Mitigation; TSX disabled
```

Part(ii)

Continuing from (d) in Part(i),

Lets convert c_n to an 2D array to reduce the number of parameters in function and more control over structure. Then we will swap this 2D array. So that we get our access pattern in row-major.
Total time = 169.982355 seconds, Speedup : 2.5x (200619-prob4-v2.c)