1. PB1

a) Performance (with command line argument 100,000,000):

	-O2	-O3
Avg	162.374	131.472
Min	157.597	126.963

Performance (with command line argument 10,000,000):

	-O2	-O3
Avg	16.656	14.572
Min	15.770	12.827

Note: all performance test is conducted 10 times to find avg and min.

b) I'm using a VM on my laptop with a 64-bit Ubuntu system. With that said, my laptop has an intel i7 CPU which is a family of x86 processors. It is running at 2592 MHz.

c) Loop Unrolling:

```c
void do_loops(int *a, int *b, int *c, int N)
{
  int i;
  for (i=N-1; i>=1; i-=9) {
    a[i] = a[i] + 1;
    a[i-1] = a[i-1] + 1;
    a[i-2] = a[i-2] + 1;
    a[i-3] = a[i-3] + 1;
    a[i-4] = a[i-4] + 1;
    a[i-5] = a[i-5] + 1;
    a[i-6] = a[i-6] + 1;
    a[i-7] = a[i-7] + 1;
    a[i-8] = a[i-8] + 1;
  }
  for (i=1; i<N; i+=9) {
    b[i] = a[i+1] + 3;
    b[i+1] = a[i+2] + 3;
    b[i+2] = a[i+3] + 3;
    b[i+3] = a[i+4] + 3;
    b[i+4] = a[i+5] + 3;
    b[i+5] = a[i+6] + 3;
    b[i+6] = a[i+7] + 3;
    b[i+7] = a[i+8] + 3;
    b[i+8] = a[i+9] + 3;
  }
```

```
  for (i=1; i<N; i+=9) {
    c[i] = b[i-1] + 2;
    c[i+1] = b[i] + 2;
    c[i+2] = b[i+1] + 2;
    c[i+3] = b[i+2] + 2;
    c[i+4] = b[i+3] + 2;
    c[i+5] = b[i+4] + 2;
    c[i+6] = b[i+5] + 2;
    c[i+7] = b[i+6] + 2;
    c[i+8] = b[i+7] + 2;
  }
}
```

Performance (with command line argument 100,000,000):

	-O2	-O3
Avg	154.482	143.259
Min	147.013	135.322

Performance (with command line argument 10,000,000):

	-O2	-O3
Avg	16.182	14.274
Min	14.419	13.426

Assembly code:

First for loop only, with -O2 compile option after loop unrolling:

```
0000000000001470 <do_loops>:
    1470:   f3 0f 1e fa     endbr64
    1474:   44 8d 41 ff     lea    -0x1(%rcx),%r8d
    1478:   45 85 c0        test   %r8d,%r8d
    147b:   7e 3b           jle    14b8 <do_loops+0x48>
    147d:   49 63 c0        movslq %r8d,%rax
    1480:   48 8d 04 87     lea    (%rdi,%rax,4),%rax
    1484:   0f 1f 40 00     nopl   0x0(%rax)
    1488:   41 83 e8 09     sub    $0x9,%r8d
    148c:   83 00 01        addl   $0x1,(%rax)
    148f:   83 40 fc 01     addl   $0x1,-0x4(%rax)
    1493:   83 40 f8 01     addl   $0x1,-0x8(%rax)
    1497:   83 40 f4 01     addl   $0x1,-0xc(%rax)
    149b:   83 40 f0 01     addl   $0x1,-0x10(%rax)
    149f:   83 40 ec 01     addl   $0x1,-0x14(%rax)
    14a3:   83 40 e8 01     addl   $0x1,-0x18(%rax)
    14a7:   83 40 e4 01     addl   $0x1,-0x1c(%rax)
```

14ab: 83 40 e0 01 addl $0x1,-0x20(%rax)

We can see that with loop unrolling, we have increased the portion of assembly code that are doing the operation, with respect to having a large portion of code taken up by loop management. The result also adheres to this observation, that is, when using -O2 compile option, we can see a significant performance improvement, which matches my expectation. For -O3 option, I am also expecting a minor improvement, but the result is a little worse compared to the original code. The reason could be that we have added this excessive code that could undermine the compiler's ability to optimize.

Loop Fusion:

```c
void do_loops(int *a, int *b, int *c, int N)
{
  int i;
  for (i=N-1; i>=1; i--) {
    a[i] = a[i] + 1;
  }
  for (i=1; i<N; i++) {
    b[i] = a[i+1] + 3;
    c[i] = b[i-1] + 2;
  }
}
```

Performance (with command line argument 100,000,000):

	-O2	-O3
Avg	160.324	171.656
Min	153.723	166.890

Performance (with command line argument 10,000,000):

	-O2	-O3
Avg	18.577	17.345
Min	15.209	16.800

Assembly Code:

0000000000001470 <do_loops>:

1470: f3 0f 1e fa endbr64

1474: 8d 41 ff lea -0x1(%rcx),%eax

1477: 85 c0 test %eax,%eax

1479: 7e 11 jle 148c <do_loops+0x1c>

147b: 48 98 cltq

147d: 0f 1f 00 nopl (%rax)

1480: 83 04 87 01 addl $0x1,(%rdi,%rax,4)

```
1484:   48 83 e8 01        sub    $0x1,%rax
1488:   85 c0              test   %eax,%eax
148a:   7f f4              jg     1480 <do_loops+0x10>
148c:   83 f9 01           cmp    $0x1,%ecx
148f:   7e 2c              jle    14bd <do_loops+0x4d>
1491:   44 8d 41 fe        lea    -0x2(%rcx),%r8d
1495:   b8 01 00 00 00     mov    $0x1,%eax
149a:   49 83 c0 02        add    $0x2,%r8
149e:   66 90              xchg   %ax,%ax
14a0:   8b 4c 87 04        mov    0x4(%rdi,%rax,4),%ecx
14a4:   83 c1 03           add    $0x3,%ecx
14a7:   89 0c 86           mov    %ecx,(%rsi,%rax,4)
14aa:   8b 4c 86 fc        mov    -0x4(%rsi,%rax,4),%ecx
14ae:   83 c1 02           add    $0x2,%ecx
14b1:   89 0c 82           mov    %ecx,(%rdx,%rax,4)
14b4:   48 83 c0 01        add    $0x1,%rax
14b8:   4c 39 c0           cmp    %r8,%rax
14bb:   75 e3              jne    14a0 <do_loops+0x30>
14bd:   c3                 retq
14be:   66 90              xchg   %ax,%ax
```

Again, with loop fusion, with the reduced overhead of loop management instructions, the code runs faster under -O2 compilation option, which met my expectation. However, again, for -O3 option, the code is a lot slower. This could be that loop fusion is not the ideal way of optimization and has in turn make the compilation harder to optimize.

Loop Strip Mining:

```c
void do_loops(int *a, int *b, int *c, int N)
{
  int i,j;
  for (i=N-1; i>=1; i-=1000) {
    for (j=i; j>(i-1000);--j)
    a[j] = a[j] + 1;
  }
  for (i=1; i<N; i+=1000) {
    for (j=i; j<(i+1000); j++){
      b[j] = a[j+1] + 3;
      c[j] = b[j-1] + 2;
    }
  }
```

```
14. }
```

Performance (with command line argument 100,000,000):

	-O2	-O3
Avg	154.485	167.668
Min	148.214	164.484

Performance (with command line argument 10,000,000):

	-O2	-O3
Avg	15.937	17.933
Min	14.907	16.385

```
0000000000001470 <do_loops>:
    1470:   f3 0f 1e fa             endbr64
    1474:   41 89 ca                mov    %ecx,%r10d
    1477:   8d 49 ff                lea    -0x1(%rcx),%ecx
    147a:   49 89 d0                mov    %rdx,%r8
    147d:   85 c9                   test   %ecx,%ecx
    147f:   7e 34                   jle    14b5 <do_loops+0x45>
    1481:   48 63 c1                movslq %ecx,%rax
    1484:   48 8d 94 87 60 f0 ff    lea    -0xfa0(%rdi,%rax,4),%rdx
    148b:   ff
    148c:   48 8d 82 a0 0f 00 00    lea    0xfa0(%rdx),%rax
    1493:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
    1498:   83 00 01                addl   $0x1,(%rax)
    149b:   48 83 e8 04             sub    $0x4,%rax
    149f:   48 39 c2                cmp    %rax,%rdx
    14a2:   75 f4                   jne    1498 <do_loops+0x28>
    14a4:   81 e9 e8 03 00 00       sub    $0x3e8,%ecx
    14aa:   48 81 ea a0 0f 00 00    sub    $0xfa0,%rdx
    14b1:   85 c9                   test   %ecx,%ecx
    14b3:   7f d7                   jg     148c <do_loops+0x1c>
    14b5:   b9 a4 0f 00 00          mov    $0xfa4,%ecx
    14ba:   41 b9 01 00 00 00       mov    $0x1,%r9d
    14c0:   41 83 fa 01             cmp    $0x1,%r10d
    14c4:   7e 4c                   jle    1512 <do_loops+0xa2>
    14c6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    14cd:   00 00 00
    14d0:   48 8d 81 60 f0 ff ff    lea    -0xfa0(%rcx),%rax
    14d7:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
    14de:   00 00
    14e0:   8b 54 07 04             mov    0x4(%rdi,%rax,1),%edx
```

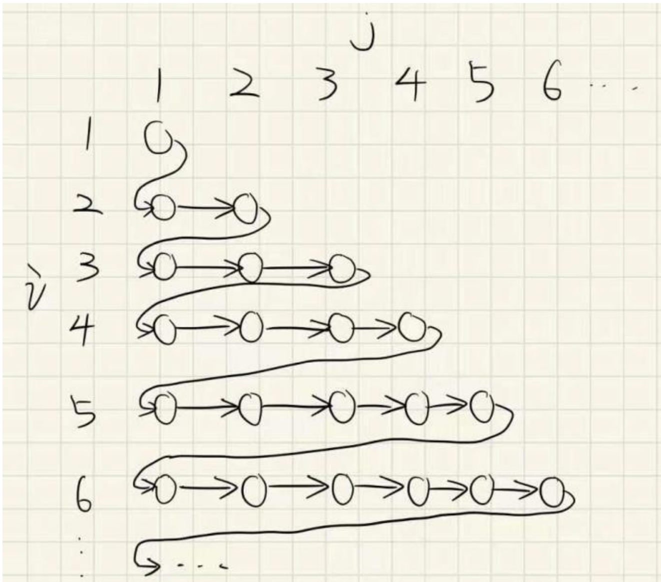
```
14e4:    83 c2 03                add    $0x3,%edx
14e7:    89 14 06                mov    %edx,(%rsi,%rax,1)
14ea:    8b 54 06 fc             mov    -0x4(%rsi,%rax,1),%edx
14ee:    83 c2 02                add    $0x2,%edx
14f1:    41 89 14 00             mov    %edx,(%r8,%rax,1)
14f5:    48 83 c0 04             add    $0x4,%rax
14f9:    48 39 c1                cmp    %rax,%rcx
14fc:    75 e2                   jne    14e0 <do_loops+0x70>
14fe:    41 81 c1 e8 03 00 00    add    $0x3e8,%r9d
1505:    48 81 c1 a0 0f 00 00    add    $0xfa0,%rcx
150c:    45 39 ca                cmp    %r9d,%r10d
150f:    7f bf                   jg     14d0 <do_loops+0x60>
1511:    c3                      retq
1512:    c3                      retq
1513:    66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
151a:    00 00 00
151d:    0f 1f 00                nopl   (%rax)
```

We can see that loop strip mining is not making much improvement based on the previous loop fusion code. Again, the code runs faster with -O2 compile option than -O3 compile option, which could be caused by the highly unrolled code, making it hard to optimize. Overall, the performance is not out of expectation.

d) Comparing compiler performance vs. hand-tune code

Based on different method across the board, we can easily beat or at-least come close when using -O2 compiling option. Loop unrolling yield the best overall performance, but the runtime is still slower than -O3 compiled original code by a small margin.

2. PB2
   a) ITG



b) Dependencies

① S1[i,j] ==> AS1[i+1,j-1] (loop-carried)

② S1[i,j] ==> AS2[i,j] (loop-independent)

③ S1[i,j] ==> TS2[i+1,j+1] (loop-carried)

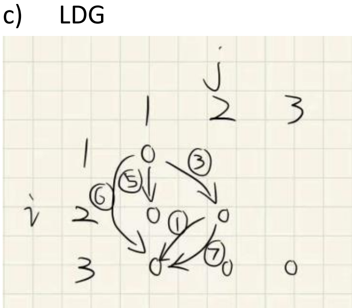
④ S1[i,j] ==> TS3[i,j] (loop-independent)

⑤ S3[i,j] ==> TS1[i+1,j] (loop-carried)

⑥ S2[i,j] ==> TS3[i+2,j] (loop-carried)

⑦ S4[i,j] ==> TS4[i+1][j-1] (loop-carried)

c) LDG



3. PB3

a) Performance without/with function inline (with command line argument 100,000,000):

	Without function inline	With function inline
Avg	338.173	137.466

Min	328.134	119.489

b) Assembly code without function inline (portion of main function captured between two gettimeofday call):

```
13e8:   e8 e3 fd ff ff          callq  11d0 <gettimeofday@plt>
13ed:   4c 8b 4c 24 50          mov    0x50(%rsp),%r9
13f2:   4c 8b 44 24 30          mov    0x30(%rsp),%r8
13f7:   31 d2                   xor    %edx,%edx
13f9:   48 8b 4c 24 70          mov    0x70(%rsp),%rcx
13fe:   66 90                   xchg   %ax,%ax
1400:   41 8b 34 91             mov    (%r9,%rdx,4),%esi
1404:   41 8b 3c 90             mov    (%r8,%rdx,4),%edi
1408:   e8 33 04 00 00          callq  1840 <_Z3addii>
140d:   89 04 91                mov    %eax,(%rcx,%rdx,4)
1410:   48 89 d0                mov    %rdx,%rax
1413:   48 83 c2 01             add    $0x1,%rdx
1417:   4c 39 e8                cmp    %r13,%rax
141a:   75 e4                   jne    1400 <main+0x1c0>
141c:   31 f6                   xor    %esi,%esi
141e:   4c 89 e7                mov    %r12,%rdi
1421:   e8 aa fd ff ff          callq  11d0 <gettimeofday@plt>
```

Assembly code with function inline (portion of main function captured between two gettimeofday call):

```
13e5:   e8 e6 fd ff ff          callq  11d0 <gettimeofday@plt>
13ea:   48 8b 4c 24 50          mov    0x50(%rsp),%rcx
13ef:   48 8b 54 24 70          mov    0x70(%rsp),%rdx
13f4:   48 8b 74 24 30          mov    0x30(%rsp),%rsi
13f9:   48 8d 41 0f             lea    0xf(%rcx),%rax
13fd:   48 29 d0                sub    %rdx,%rax
1400:   48 83 f8 1e             cmp    $0x1e,%rax
1404:   48 8d 46 0f             lea    0xf(%rsi),%rax
1408:   40 0f 97 c7             seta   %dil
140c:   48 29 d0                sub    %rdx,%rax
140f:   48 83 f8 1e             cmp    $0x1e,%rax
1413:   0f 97 c0                seta   %al
1416:   40 84 c7                test   %al,%dil
1419:   0f 84 72 02 00 00       je     1691 <main+0x451>
141f:   83 fd 02                cmp    $0x2,%ebp
1422:   0f 86 69 02 00 00       jbe    1691 <main+0x451>
1428:   89 df                   mov    %ebx,%edi
142a:   31 c0                   xor    %eax,%eax
142c:   c1 ef 02                shr    $0x2,%edi
142f:   48 c1 e7 04             shl    $0x4,%rdi
```

```
1433:    0f 1f 44 00 00       nopl    0x0(%rax,%rax,1)
1438:    f3 0f 6f 04 01       movdqu (%rcx,%rax,1),%xmm0
143d:    f3 0f 6f 1c 06       movdqu (%rsi,%rax,1),%xmm3
1442:    66 0f fe c3          paddd   %xmm3,%xmm0
1446:    0f 11 04 02          movups %xmm0,(%rdx,%rax,1)
144a:    48 83 c0 10          add     $0x10,%rax
144e:    48 39 f8             cmp     %rdi,%rax
1451:    75 e5                jne     1438 <main+0x1f8>
1453:    89 d8                mov     %ebx,%eax
1455:    83 e0 fc             and     $0xfffffffc,%eax
1458:    f6 c3 03             test    $0x3,%bl
145b:    74 37                je      1494 <main+0x254>
145d:    48 63 f8             movslq %eax,%rdi
1460:    44 8b 04 b9          mov     (%rcx,%rdi,4),%r8d
1464:    44 03 04 be          add     (%rsi,%rdi,4),%r8d
1468:    44 89 04 ba          mov     %r8d,(%rdx,%rdi,4)
146c:    8d 78 01             lea     0x1(%rax),%edi
146f:    39 fb                cmp     %edi,%ebx
1471:    7e 21                jle     1494 <main+0x254>
1473:    48 63 ff             movslq %edi,%rdi
1476:    83 c0 02             add     $0x2,%eax
1479:    44 8b 04 be          mov     (%rsi,%rdi,4),%r8d
147d:    44 03 04 b9          add     (%rcx,%rdi,4),%r8d
1481:    44 89 04 ba          mov     %r8d,(%rdx,%rdi,4)
1485:    39 c3                cmp     %eax,%ebx
1487:    7e 0b                jle     1494 <main+0x254>
1489:    48 98                cltq
148b:    8b 34 86             mov     (%rsi,%rax,4),%esi
148e:    03 34 81             add     (%rcx,%rax,4),%esi
1491:    89 34 82             mov     %esi,(%rdx,%rax,4)
1494:    31 f6                xor     %esi,%esi
1496:    4c 89 e7             mov     %r12,%rdi
1499:    e8 32 fd ff ff       callq   11d0 <gettimeofday@plt>
```

We can see from the above assembly code snippet, inline eliminate function calls to the add function and direct did the addition in the main function, which, should eliminate the function call/return control code, eventually reduce the runtime.

c) The performance adheres to my expectation. Since function calls requires extra instruction to control the call/return of the function, which involves processes like reading variable into registers, creating special registers like stack pointer. All these takes time. Although in-lining increases code segment size, for such a simple function, the trade-off is clear.

d) Performance of original code (with command line argument 100,000,000):

Avg	62.525
Min	59.548

The result of the original code is even better than the code after function in-lining. Which indicate the compiler has made further optimization based on function in-lining, but I haven't figure out why there is a stable improvement of performance with the original code.

4. PB4

a) Invariant Hoisting (Pull non-loop-dependent calculations out of loop)

```
int a[N][4];
int rand_number = rand();
int threshold = 2.0 * rand_number;
for (i=0; i<4; i++){
  for (j=0; j<N; j++){
    if (threshold < 4){
      sum = sum + a[j][i];
    }
    else{
      sum = sum + a[j][i] + 1;
    }
  }
}
```

b) Loop Un-switching (move a conditional expression outside of a loop, replicate loop body inside of each conditional block):

```
int a[N][4];
int rand_number = rand();
int threshold = 2.0 * rand_number;
if (threshold < 4){
    for (i=0; i<4; i++){
        for (j=0; j<N; j++){
            sum = sum + a[j][i];
        }
    }
}
else {
    for (i=0; i<4; i++){
        for (j=0; j<N; j++){
```

```
            sum = sum + a[j][i] + 1;
        }
    }
}
```

c) Loop Interchange (switch the positions of one loop that is tightly nested within another loop):

```
int a[N][4];
int rand_number = rand();
int threshold = 2.0 * rand_number;
if (threshold < 4){
    for (j=0; j<N; j++){
        for (i=0; i<4; i++){
            sum = sum + a[j][i];
        }
    }
}
else {
    for (j=0; j<N; j++){
        for (i=0; i<4; i++){
            sum = sum + a[j][i] + 1;
        }
    }
}
```

d) Loop Unroll and Jam (Partially unroll one or more loops higher in the loop nest than the innermost loop, then jam resulting loops back together):

```
int a[N][4];
int rand_number = rand();
int threshold = 2.0 * rand_number;
if (threshold < 4){
    for (j=0; j<N; j+=2){
        for (i=0; i<4; i++){
            sum = sum + a[j][i];
            sum = sum + a[j+1][i];
        }
    }
}
else {
```

```
    for (j=0; j<N; j+=2){
        for (i=0; i<4; i++){
            sum = sum + a[j][i] + 1;
            sum = sum + a[j+1][i] + 1;
        }
    }
}
```

e) Loop Unrolling (Combine multiple instances of the loop body, making reduction to the loop iteration count):

```
int a[N][4];
int rand_number = rand();
int threshold = 2.0 * rand_number;
if (threshold < 4){
    for (j=0; j<N; j+=2){
        sum = sum + a[j][i];
        sum = sum + a[j+1][i];
        sum = sum + a[j][i+1];
        sum = sum + a[j+1][i+1];
        sum = sum + a[j][i+2];
        sum = sum + a[j+1][i+2];
        sum = sum + a[j][i+3];
        sum = sum + a[j+1][i+3];
    }
}
else {
    for (j=0; j<N; j+=2){
        sum = sum + a[j][i] + 1;
        sum = sum + a[j+1][i] + 1;
        sum = sum + a[j][i+1] + 1;
        sum = sum + a[j+1][i+1] + 1;
        sum = sum + a[j][i+2] + 1;
        sum = sum + a[j+1][i+2] + 1;
        sum = sum + a[j][i+3] + 1;
        sum = sum + a[j+1][i+3] + 1;
    }
}
```

5. PB5
   a) It is not safe.
      For the original code, we have a loop-carried output dependency S1[i]➔OS3[i-1] and a loop-carried dependency S2[i]➔AS3[i+1](excluding dependencies that are not affected).
      However, after loop fusion, we have an altered loop-carried output dependency S3[i]➔OS2[i+1] and a loop-carried dependency S3[i]➔TS2[i-1]. Which in other words, a reverse in dependency.
   b) It is not safe.
      The current dependency of the loop is S1[i,j]➔AS1[i,j], which violate the observation that outermost loop shouldn’t carry a dependency with i<i’ and j>j’, making it unsafe.
   c) Safe