ECE 565 homework 1

Js895 **Junqi Sun**

Question1 :

(a)

Exe time(milliseconds)	Optimization level		Original output
Cmd line argument	O2	O3	No optimization
10,000,000	22.885	18.366	77.973
100,000,000	227.108	184.820	776.493

All result are minimum value from 5 tests

Original O2 assembly code:

```
for (i=N-1; i>=1; i--) {
  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)
  a[i] = a[i] + 1;
  1480:       83 04 87 01           addl   $0x1,(%rdi,%rax,4)
for (i=N-1; i>=1; i--) {
  1484:       48 83 e8 01           sub    $0x1,%rax
  1488:       85 c0                 test   %eax,%eax
  148a:       7f f4                 jg     1480 <do_loops+0x10>
```

```
for (i=1; i<N; i++) {
  148c:       83 f9 01              cmp    $0x1,%ecx
  148f:       7e 3d                 jle    14ce <do_loops+0x5e>
  1491:       44 8d 41 fe           lea    -0x2(%rcx),%r8d
  1495:       b8 02 00 00 00        mov    $0x2,%eax
  149a:       4d 8d 48 03           lea    0x3(%r8),%r9
  149e:       66 90                 xchg   %ax,%ax
  b[i] = a[i+1] + 3;
  14a0:       8b 0c 87              mov    (%rdi,%rax,4),%ecx
  14a3:       83 c1 03              add    $0x3,%ecx
  14a6:       89 4c 86 fc           mov    %ecx,-0x4(%rsi,%rax,4)
```

```
  for (i=1; i<N; i++) {
    14aa:       48 83 c0 01             add    $0x1,%rax
    14ae:       49 39 c1                cmp    %rax,%r9
    14b1:       75 ed                   jne    14a0 <do_loops+0x30>
    14b3:       31 c0                   xor    %eax,%eax
    14b5:       0f 1f 00                nopl   (%rax)
    c[i] = b[i-1] + 2;
    14b8:       8b 3c 86                mov    (%rsi,%rax,4),%edi
    14bb:       8d 4f 02                lea    0x2(%rdi),%ecx
    14be:       89 4c 82 04             mov    %ecx,0x4(%rdx,%rax,4)
  for (i=1; i<N; i++) {
    14c2:       48 89 c1                mov    %rax,%rcx
    14c5:       48 83 c0 01             add    $0x1,%rax
    14c9:       4c 39 c1                cmp    %r8,%rcx
    14cc:       75 ea                   jne    14b8 <do_loops+0x48>
}
```

(b)

Processor architecture: x86_64
CPU frequency: 2.3GHZ
OS type: Linux
VM: Ubuntu 20.04.1 LTS

(c)

Optimization 1: loop unrolling

Source code:

```
// loop unrolling
int i;
for (i=N-1; i>=1; i-=4) {
  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;
}

for (i=1; i<N; i+=4) {
  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;
}

for (i=1; i<N; i+=4) {
  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;
}
```

Performance result:

Exe time(milliseconds)	Optimization level	
Cmd line argument	O2	O3
10,000,000	21.402	19.751
100,000,000	208.817	199.864

Assembly Code of O2:

```
for (i=N-1; i>=1; i-=4) {
  1474:       8d 41 ff                lea    -0x1(%rcx),%eax
  1477:       85 c0                   test   %eax,%eax
  1479:       7e 3d                   jle    14b8 <do_loops+0x48>
  147b:       48 98                   cltq
  147d:       44 8d 49 fe             lea    -0x2(%rcx),%r9d
  1481:       4c 8d 04 85 00 00 00    lea    0x0(,%rax,4),%r8
  1488:       00
  1489:       41 c1 e9 02             shr    $0x2,%r9d
  148d:       4a 8d 04 07             lea    (%rdi,%r8,1),%rax
  1491:       49 c1 e1 04             shl    $0x4,%r9
  1495:       4e 8d 44 07 f0          lea    -0x10(%rdi,%r8,1),%r8
  149a:       4d 29 c8                sub    %r9,%r8
  149d:       0f 1f 00                nopl   (%rax)
  a[i] = a[i] + 1;
  14a0:       83 00 01                addl   $0x1,(%rax)
  a[i-1] = a[i-1] + 1;
  14a3:       83 40 fc 01             addl   $0x1,-0x4(%rax)
  a[i-2] = a[i-2] + 1;
  14a7:       83 40 f8 01             addl   $0x1,-0x8(%rax)
  a[i-3] = a[i-3] + 1;
  14ab:       83 40 f4 01             addl   $0x1,-0xc(%rax)
for (i=N-1; i>=1; i-=4) {
  14af:       48 83 e8 10             sub    $0x10,%rax
  14b3:       4c 39 c0                cmp    %r8,%rax
  14b6:       75 e8                   jne    14a0 <do_loops+0x30>
```

```
for (i=1; i<N; i+=4) {
  14b8:       83 f9 01                cmp    $0x1,%ecx
  14bb:       0f 8e 8f 00 00 00       jle    1550 <do_loops+0xe0>
  14c1:       44 8d 49 fe             lea    -0x2(%rcx),%r9d
  14c5:       48 8d 47 08             lea    0x8(%rdi),%rax
  14c9:       44 89 c9                mov    %r9d,%ecx
  14cc:       4c 8d 46 04             lea    0x4(%rsi),%r8
  14d0:       c1 e9 02                shr    $0x2,%ecx
  14d3:       41 89 c9                mov    %ecx,%r9d
  14d6:       49 c1 e1 04             shl    $0x4,%r9
  14da:       4a 8d 7c 0f 18          lea    0x18(%rdi,%r9,1),%rdi
  14df:       90                      nop
  b[i] = a[i+1] + 3;
  14e0:       8b 08                   mov    (%rax),%ecx
  14e2:       48 83 c0 10             add    $0x10,%rax
  14e6:       49 83 c0 10             add    $0x10,%r8
  14ea:       83 c1 03                add    $0x3,%ecx
  14ed:       41 89 48 f0             mov    %ecx,-0x10(%r8)
  b[i+1] = a[i+2] + 3;
  14f1:       8b 48 f4                mov    -0xc(%rax),%ecx
  14f4:       83 c1 03                add    $0x3,%ecx
  14f7:       41 89 48 f4             mov    %ecx,-0xc(%r8)
  b[i+2] = a[i+3] + 3;
  14fb:       8b 48 f8                mov    -0x8(%rax),%ecx
  14fe:       83 c1 03                add    $0x3,%ecx
  1501:       41 89 48 f8             mov    %ecx,-0x8(%r8)
  b[i+3] = a[i+4] + 3;
  1505:       8b 48 fc                mov    -0x4(%rax),%ecx
  1508:       83 c1 03                add    $0x3,%ecx
  150b:       41 89 48 fc             mov    %ecx,-0x4(%r8)
```

```
  for (i=1; i<N; i+=4) {
    150f:       48 39 f8                cmp    %rdi,%rax
    1512:       75 cc                   jne    14e0 <do_loops+0x70>
    1514:       48 83 c2 04             add    $0x4,%rdx
    1518:       4a 8d 4c 0e 10          lea    0x10(%rsi,%r9,1),%rcx
    151d:       0f 1f 00                nopl   (%rax)
    c[i] = b[i-1] + 2;
    1520:       8b 06                   mov    (%rsi),%eax
    1522:       48 83 c6 10             add    $0x10,%rsi
    1526:       48 83 c2 10             add    $0x10,%rdx
    152a:       83 c0 02                add    $0x2,%eax
    152d:       89 42 f0                mov    %eax,-0x10(%rdx)
    c[i+1] = b[i] + 2;
    1530:       8b 46 f4                mov    -0xc(%rsi),%eax
    1533:       83 c0 02                add    $0x2,%eax
    1536:       89 42 f4                mov    %eax,-0xc(%rdx)
    c[i+2] = b[i+1] + 2;
    1539:       8b 46 f8                mov    -0x8(%rsi),%eax
    153c:       83 c0 02                add    $0x2,%eax
    153f:       89 42 f8                mov    %eax,-0x8(%rdx)
    c[i+3] = b[i+2] + 2;
    1542:       8b 46 fc                mov    -0x4(%rsi),%eax
    1545:       83 c0 02                add    $0x2,%eax
    1548:       89 42 fc                mov    %eax,-0x4(%rdx)
  for (i=1; i<N; i+=4) {
    154b:       48 39 ce                cmp    %rcx,%rsi
    154e:       75 d0                   jne    1520 <do_loops+0xb0>
}
```

Analysis:
Compare to the original code with O2, O3, the unrolling version of O2 is faster than original while O3 is a bit slower.

When refering to assembly code: In the first loop in **original code**, we use %rax as index i, and do **addl $0x1, (%rdi, %rax, 4) every time in loop.** While in **unrolling loop**, this step changes to **addl $0x1, (%rax), addl $0x1, -0x4(%rax), addl $0x1, -0x8(%rax), addl $0x1, -0xc(%rax)**. I think this adjustment here means we don't need to fetch for %rdi every time, we just need to add instant number which is faster.

In order to achieve that, compiler did more preparation step when first enter the loop, which load array address information(%rdi) into %rax before the first operation, thus save the time. The following loop perform similar methods to optimize.

Loop unrolling also expose additional ILP, and reduce instruction count to improve performance.

Optimization 2: loop fusion

Source code:

```
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 result:

Exe time(milliseconds)	Optimization level	
Cmd line argument	O2	O3
10,000,000	22.631	22.200
100,000,000	213.774	222.919

Assembly Code of O2:

```
   1470:       f3 0f 1e fa             endbr64
  for (i=N-1; i>=1; i--) {
   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)
   a[i] = a[i] + 1;
   1480:       83 04 87 01             addl   $0x1,(%rdi,%rax,4)
  for (i=N-1; i>=1; i--) {
   1484:       48 83 e8 01             sub    $0x1,%rax
   1488:       85 c0                   test   %eax,%eax
   148a:       7f f4                   jg     1480 <do_loops+0x10>
 for (i=1; i<N; i++) {
   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
   b[i] = a[i+1] + 3;
   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)
   c[i] = b[i-1] + 2;
   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)
 for (i=1; i<N; i++) {
   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
```

Analysis:

Compare to the original code with O2, O3, the O2 is a little bit faster than original one, while O3 is still slower than before.

When referring to assembly code: In the second and third loop (since the first loop doesn't change) in original code: we need to do the preparation work (from **cmp $0x1, %ecx** to **xchg %ax, %ax**) twice (one for each loop initialization). For operation on b[i] and c[i], original code use %ecx to store value for b, %edi to store value for c. And use **lea 0x2(%rdi), %ecx** to plus 2, which need an additional read from %rdi.

In fusion code, we only initialize once to access the address of array[a], that we reduce overhead of loop management and improve data locality. Also we could reuse the %ecx to store value for b and c, thus cut down additional register usage.

Optimization 3: loop peeling

Source code:

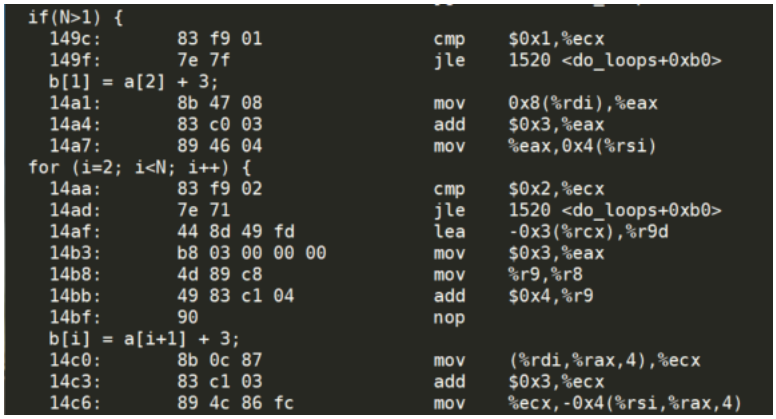
```
int i;
if(N>=3) {
  a[N-1] = a[N-1] + 1;
}
for (i=N-2; i>=1; i--) {
  a[i] = a[i] + 1;
}
if(N>1) {
  b[1] = a[2] + 3;
}
for (i=2; i<N; i++) {
  b[i] = a[i+1] + 3;
}
if(N>1) {
  c[1] = b[0] + 2;
}
for (i=2; i<N; i++) {
  c[i] = b[i-1] + 2;
  }
```

Performance result:

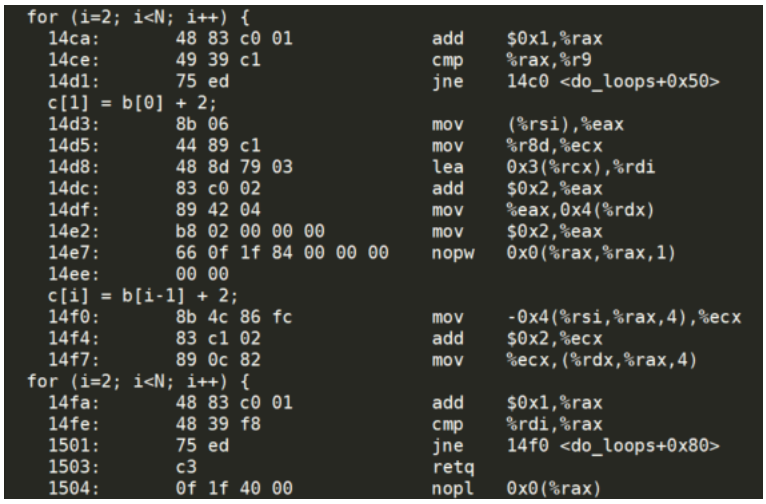
Exe time(milliseconds)	Optimization level	
Cmd line argument	O2	O3
10,000,000	22.629	18.912
100,000,000	225.357	187.003

Assembly Code of O2:

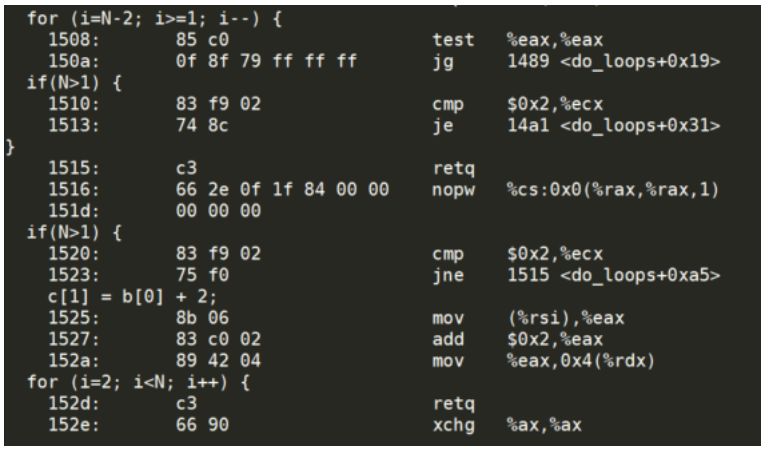
```
if(N>=3) {
  1474:       8d 41 fe                lea    -0x2(%rcx),%eax
  1477:       83 f9 02                cmp    $0x2,%ecx
  147a:       0f 8e 88 00 00 00       jle    1508 <do_loops+0x98>
  a[N-1] = a[N-1] + 1;
  1480:       4c 63 c1                movslq %ecx,%r8
  1483:       42 83 44 87 fc 01       addl   $0x1,-0x4(%rdi,%r8,4)
for (i=N-2; i>=1; i--) {
  1489:       48 98                   cltq
  148b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  a[i] = a[i] + 1;
  1490:       83 04 87 01             addl   $0x1,(%rdi,%rax,4)
for (i=N-2; i>=1; i--) {
  1494:       48 83 e8 01             sub    $0x1,%rax
  1498:       85 c0                   test   %eax,%eax
  149a:       7f f4                   jg     1490 <do_loops+0x20>
```



```
  if(N>1) {
    149c:       83 f9 01                cmp    $0x1,%ecx
    149f:       7e 7f                   jle    1520 <do_loops+0xb0>
    b[1] = a[2] + 3;
    14a1:       8b 47 08                mov    0x8(%rdi),%eax
    14a4:       83 c0 03                add    $0x3,%eax
    14a7:       89 46 04                mov    %eax,0x4(%rsi)
  for (i=2; i<N; i++) {
    14aa:       83 f9 02                cmp    $0x2,%ecx
    14ad:       7e 71                   jle    1520 <do_loops+0xb0>
    14af:       44 8d 49 fd             lea    -0x3(%rcx),%r9d
    14b3:       b8 03 00 00 00          mov    $0x3,%eax
    14b8:       4d 89 c8                mov    %r9,%r8
    14bb:       49 83 c1 04             add    $0x4,%r9
    14bf:       90                      nop
    b[i] = a[i+1] + 3;
    14c0:       8b 0c 87                mov    (%rdi,%rax,4),%ecx
    14c3:       83 c1 03                add    $0x3,%ecx
    14c6:       89 4c 86 fc             mov    %ecx,-0x4(%rsi,%rax,4)
```



```
  for (i=2; i<N; i++) {
    14ca:       48 83 c0 01             add    $0x1,%rax
    14ce:       49 39 c1                cmp    %rax,%r9
    14d1:       75 ed                   jne    14c0 <do_loops+0x50>
    c[1] = b[0] + 2;
    14d3:       8b 06                   mov    (%rsi),%eax
    14d5:       44 89 c1                mov    %r8d,%ecx
    14d8:       48 8d 79 03             lea    0x3(%rcx),%rdi
    14dc:       83 c0 02                add    $0x2,%eax
    14df:       89 42 04                mov    %eax,0x4(%rdx)
    14e2:       b8 02 00 00 00          mov    $0x2,%eax
    14e7:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
    14ee:       00 00
    c[i] = b[i-1] + 2;
    14f0:       8b 4c 86 fc             mov    -0x4(%rsi,%rax,4),%ecx
    14f4:       83 c1 02                add    $0x2,%ecx
    14f7:       89 0c 82                mov    %ecx,(%rdx,%rax,4)
  for (i=2; i<N; i++) {
    14fa:       48 83 c0 01             add    $0x1,%rax
    14fe:       48 39 f8                cmp    %rdi,%rax
    1501:       75 ed                   jne    14f0 <do_loops+0x80>
    1503:       c3                      retq
    1504:       0f 1f 40 00             nopl   0x0(%rax)
```



```
  for (i=N-2; i>=1; i--) {
    1508:       85 c0                   test   %eax,%eax
    150a:       0f 8f 79 ff ff ff       jg     1489 <do_loops+0x19>
  if(N>1) {
    1510:       83 f9 02                cmp    $0x2,%ecx
    1513:       74 8c                   je     14a1 <do_loops+0x31>
}
    1515:       c3                      retq
    1516:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    151d:       00 00 00
  if(N>1) {
    1520:       83 f9 02                cmp    $0x2,%ecx
    1523:       75 f0                   jne    1515 <do_loops+0xa5>
    c[1] = b[0] + 2;
    1525:       8b 06                   mov    (%rsi),%eax
    1527:       83 c0 02                add    $0x2,%eax
    152a:       89 42 04                mov    %eax,0x4(%rdx)
  for (i=2; i<N; i++) {
    152d:       c3                      retq
    152e:       66 90                   xchg   %ax,%ax
```

Analysis:
Compare to the original code with O2, O3, the loop peeling version is a little bit faster than the original ones in both O2 and O3 optimization(almost the same).

When referring to assembly code: the original code initialize the address at the beginning of each loop, then implement the add operation for corresponding operations.

In loop peeling version, however, 1. It enforce mem alignment for array before loop, 2. codes are interweaved. For example, we enter the first if sentence and finish the assignment,

then the loop for a[i]. Later, the second if sentence, and loop for b[i]. However, the jump(loop judgment) sentence for b[i] operation resides in the next loop block, as a simplified way of if(). After the above section, there are code to check if we miss any circumstance due to this interweaving structure, if it does, jump back to previous section.

Optimization 4: loop reversal

Source code:

```
// loop reversal
int i;
for (i=1; i<N; i++) {
  a[i] = a[i] + 1;
}
for (i=N-1; i>=1; i--) {
  b[i] = a[i+1] + 3;
}
for (i=N-1; i>=1; i--) {
  c[i] = b[i-1] + 2;
}
```

Performance result:

Exe time(milliseconds)	Optimization level	
Cmd line argument	O2	O3
10,000,000	21.867	18.528
100,000,000	215.914	185.117

Assembly Code of O2:

```
    1470:       f3 0f 1e fa             endbr64
  for (i=1; i<N; i++) {
    1474:       8d 41 ff                lea    -0x1(%rcx),%eax
    1477:       83 f9 01                cmp    $0x1,%ecx
    147a:       7e 5c                   jle    14d8 <do_loops+0x68>
    147c:       83 e9 02                sub    $0x2,%ecx
    147f:       4c 8d 47 04             lea    0x4(%rdi),%r8
    1483:       48 8d 4c 8f 08          lea    0x8(%rdi,%rcx,4),%rcx
    1488:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    148f:       00
    a[i] = a[i] + 1;
    1490:       41 83 00 01             addl   $0x1,(%r8)
  for (i=1; i<N; i++) {
    1494:       49 83 c0 04             add    $0x4,%r8
    1498:       4c 39 c1                cmp    %r8,%rcx
    149b:       75 f3                   jne    1490 <do_loops+0x20>
    149d:       48 98                   cltq
{
    149f:       48 89 c1                mov    %rax,%rcx
    14a2:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    b[i] = a[i+1] + 3;
    14a8:       44 8b 4c 8f 04          mov    0x4(%rdi,%rcx,4),%r9d
    14ad:       45 8d 41 03             lea    0x3(%r9),%r8d
    14b1:       44 89 04 8e             mov    %r8d,(%rsi,%rcx,4)
  for (i=N-1; i>=1; i--) {
    14b5:       48 83 e9 01             sub    $0x1,%rcx
    14b9:       85 c9                   test   %ecx,%ecx
    14bb:       7f eb                   jg     14a8 <do_loops+0x38>
    14bd:       0f 1f 00                nopl   (%rax)
    c[i] = b[i-1] + 2;
    14c0:       8b 7c 86 fc             mov    -0x4(%rsi,%rax,4),%edi
    14c4:       8d 4f 02                lea    0x2(%rdi),%ecx
    14c7:       89 0c 82                mov    %ecx,(%rdx,%rax,4)
  for (i=N-1; i>=1; i--) {
    14ca:       48 83 e8 01             sub    $0x1,%rax
    14ce:       85 c0                   test   %eax,%eax
    14d0:       7f ee                   jg     14c0 <do_loops+0x50>
}
    14d2:       c3                      retq
    14d3:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  for (i=N-1; i>=1; i--) {
    14d8:       85 c0                   test   %eax,%eax
    14da:       7f c1                   jg     149d <do_loops+0x2d>
}
    14dc:       c3                      retq
    14dd:       0f 1f 00                nopl   (%rax)
```

Analysis:
Compare to the original code with O2, O3, the loop reversal version is faster than the original ones in both O2 and O3 optimization.

In this case, the original code's assembly code is very similar to the reversal one, while the reversal one may enable more optimization choices to compiler.

Optimization 5: loop strip mining

Source code:

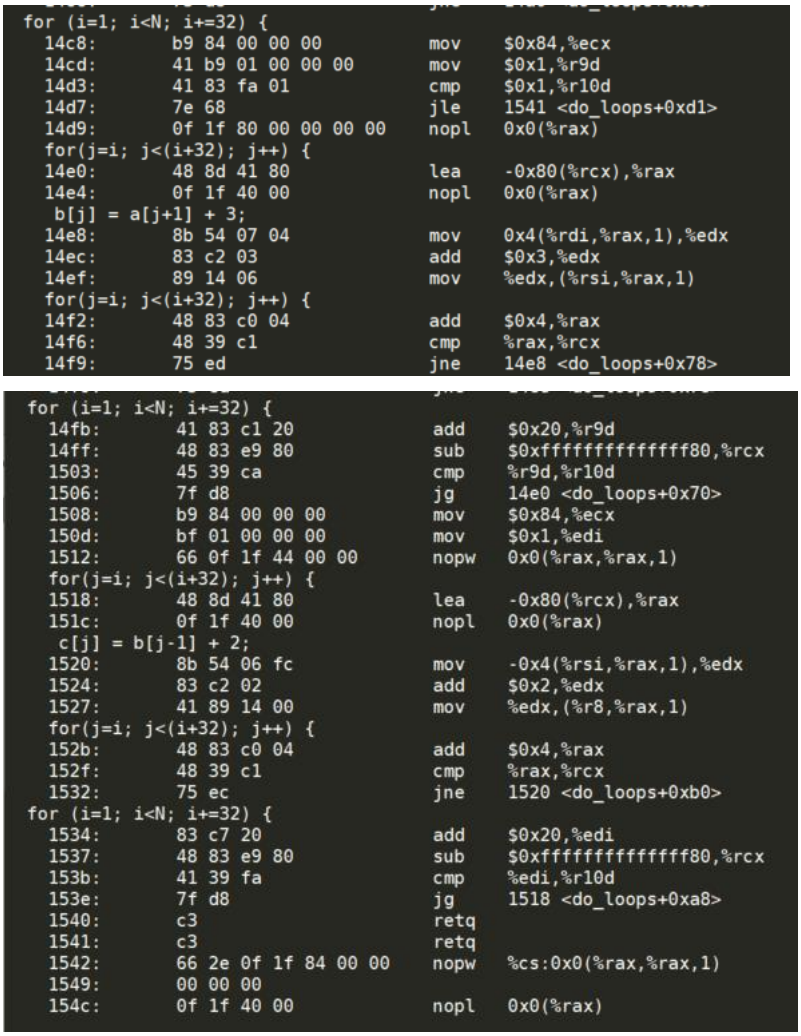
```
// loop strip mining
int i;
int j;
for (i=N-1; i>=1; i-=32) {
  for(j=i;j>(i-32);j--) {
   a[j] = a[j] + 1;
  }
}
for (i=1; i<N; i+=32) {
  for(j=i; j<(i+32); j++) {
   b[j] = a[j+1] + 3;
  }
}
for (i=1; i<N; i+=32) {
  for(j=i; j<(i+32); j++) {
   c[j] = b[j-1] + 2;
  }
}
```

Performance result:

Exe time(milliseconds)	Optimization level	
Cmd line argument	O2	O3
10,000,000	24.191	17.921
100,000,000	236.418	182.449

Assembly Code of O2:

```
  for (i=N-1; i>=1; i-=32) {
    1477:       8d 49 ff                lea    -0x1(%rcx),%ecx
{
    147a:       49 89 d0                mov    %rdx,%r8
  for (i=N-1; i>=1; i-=32) {
    147d:       85 c9                   test   %ecx,%ecx
    147f:       7e 47                   jle    14c8 <do_loops+0x58>
    1481:       48 63 c1                movslq %ecx,%rax
    1484:       45 8d 4a df             lea    -0x21(%r10),%r9d
    1488:       48 8d 54 87 80          lea    -0x80(%rdi,%rax,4),%rdx
    148d:       41 8d 42 fe             lea    -0x2(%r10),%eax
    1491:       83 e0 e0                and    $0xffffffe0,%eax
    1494:       41 29 c1                sub    %eax,%r9d
    1497:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
    149e:       00 00
    for(j=i;j>(i-32);j--) {
    14a0:       48 8d 82 80 00 00 00    lea    0x80(%rdx),%rax
    14a7:       66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
    14ae:       00 00
     a[j] = a[j] + 1;
    14b0:       83 00 01                addl   $0x1,(%rax)
    for(j=i;j>(i-32);j--) {
    14b3:       48 83 e8 04             sub    $0x4,%rax
    14b7:       48 39 c2                cmp    %rax,%rdx
    14ba:       75 f4                   jne    14b0 <do_loops+0x40>
  for (i=N-1; i>=1; i-=32) {
    14bc:       83 e9 20                sub    $0x20,%ecx
    14bf:       48 83 c2 80             add    $0xffffffffffffff80,%rdx
    14c3:       41 39 c9                cmp    %ecx,%r9d
    14c6:       75 d8                   jne    14a0 <do_loops+0x30>
```



```
for (i=1; i<N; i+=32) {
  14c8:       b9 84 00 00 00          mov    $0x84,%ecx
  14cd:       41 b9 01 00 00 00       mov    $0x1,%r9d
  14d3:       41 83 fa 01             cmp    $0x1,%r10d
  14d7:       7e 68                   jle    1541 <do_loops+0xd1>
  14d9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)
  for(j=i; j<(i+32); j++) {
  14e0:       48 8d 41 80             lea    -0x80(%rcx),%rax
  14e4:       0f 1f 40 00             nopl   0x0(%rax)
   b[j] = a[j+1] + 3;
  14e8:       8b 54 07 04             mov    0x4(%rdi,%rax,1),%edx
  14ec:       83 c2 03                add    $0x3,%edx
  14ef:       89 14 06                mov    %edx,(%rsi,%rax,1)
  for(j=i; j<(i+32); j++) {
  14f2:       48 83 c0 04             add    $0x4,%rax
  14f6:       48 39 c1                cmp    %rax,%rcx
  14f9:       75 ed                   jne    14e8 <do_loops+0x78>
```

```
for (i=1; i<N; i+=32) {
  14fb:       41 83 c1 20             add    $0x20,%r9d
  14ff:       48 83 e9 80             sub    $0xffffffffffffff80,%rcx
  1503:       45 39 ca                cmp    %r9d,%r10d
  1506:       7f d8                   jg     14e0 <do_loops+0x70>
  1508:       b9 84 00 00 00          mov    $0x84,%ecx
  150d:       bf 01 00 00 00          mov    $0x1,%edi
  1512:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
  for(j=i; j<(i+32); j++) {
  1518:       48 8d 41 80             lea    -0x80(%rcx),%rax
  151c:       0f 1f 40 00             nopl   0x0(%rax)
   c[j] = b[j-1] + 2;
  1520:       8b 54 06 fc             mov    -0x4(%rsi,%rax,1),%edx
  1524:       83 c2 02                add    $0x2,%edx
  1527:       41 89 14 00             mov    %edx,(%r8,%rax,1)
  for(j=i; j<(i+32); j++) {
  152b:       48 83 c0 04             add    $0x4,%rax
  152f:       48 39 c1                cmp    %rax,%rcx
  1532:       75 ec                   jne    1520 <do_loops+0xb0>
for (i=1; i<N; i+=32) {
  1534:       83 c7 20                add    $0x20,%edi
  1537:       48 83 e9 80             sub    $0xffffffffffffff80,%rcx
  153b:       41 39 fa                cmp    %edi,%r10d
  153e:       7f d8                   jg     1518 <do_loops+0xa8>
  1540:       c3                      retq
  1541:       c3                      retq
  1542:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  1549:       00 00 00
  154c:       0f 1f 40 00             nopl   0x0(%rax)
```

Analysis:
Compare to the original code with O2, O3, the O3 is a little bit faster than original one, while O2 is slower than before.

Since we access a block size of 32 each time, we improve the locality of this function. In strip mining code, Take a[j] = a[j] + 1 for example, the loop assembly section's code is less than original ones, which may help to improve performance. However, since there are more for() loop sentence so we need more **lea, nopw** and other command to initialize, so this may cause O2 to be slower than original ones.

Optimization 6: function inline

Source code:

```
struct timeval start_time, end_time;
gettimeofday(&start_time, NULL);
//do_loops(a, b, c, N);

for (i=N-1; i>=1; i--) {
  a[i] = a[i] + 1;
}
for (i=1; i<N; i++) {
  b[i] = a[i+1] + 3;
}
for (i=1; i<N; i++) {
  c[i] = b[i-1] + 2;
}
gettimeofday(&end_time, NULL);

for (i=0; i<N; i++) {
  sum += c[i];
}

double elapsed_us = calc_time(start_time, end_time);
double elapsed_ms = elapsed_us / 1000.0;
printf("sum=%llu\n", sum);
printf("Time=%f milliseconds\n", elapsed_ms);

return 0;
```

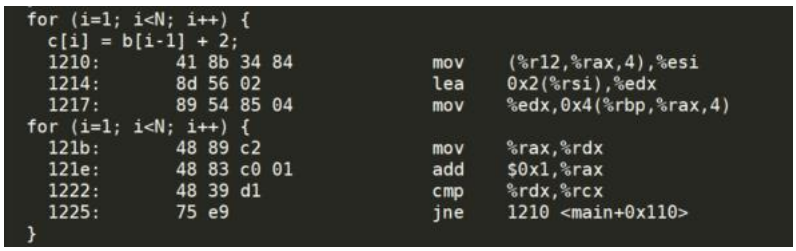
Performance result:

Exe time(milliseconds)	Optimization level	
Cmd line argument	O2	O3
10,000,000	22.987	18.427
100,000,000	224.757	185.468

Assembly Code of O2:

```
for (i=N-1; i>=1; i--) {
  11be:       41 8d 46 ff             lea    -0x1(%r14),%eax
  11c2:       85 c0                   test   %eax,%eax
  11c4:       7e 16                   jle    11dc <main+0xdc>
  11c6:       48 98                   cltq
  11c8:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
  11cf:       00
  a[i] = a[i] + 1;
  11d0:       83 04 83 01             addl   $0x1,(%rbx,%rax,4)
for (i=N-1; i>=1; i--) {
  11d4:       48 83 e8 01             sub    $0x1,%rax
  11d8:       85 c0                   test   %eax,%eax
  11da:       7f f4                   jg     11d0 <main+0xd0>
}
```

```
for (i=1; i<N; i++) {
  11dc:       41 83 fe 01             cmp    $0x1,%r14d
  11e0:       0f 8e 1f 01 00 00       jle    1305 <main+0x205>
  11e6:       41 8d 4e fe             lea    -0x2(%r14),%ecx
  11ea:       b8 02 00 00 00          mov    $0x2,%eax
  11ef:       48 8d 71 03             lea    0x3(%rcx),%rsi
  11f3:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  b[i] = a[i+1] + 3;
  11f8:       8b 3c 83                mov    (%rbx,%rax,4),%edi
  11fb:       8d 57 03                lea    0x3(%rdi),%edx
  11fe:       41 89 54 84 fc          mov    %edx,-0x4(%r12,%rax,4)
for (i=1; i<N; i++) {
  1203:       48 83 c0 01             add    $0x1,%rax
  1207:       48 39 c6                cmp    %rax,%rsi
  120a:       75 ec                   jne    11f8 <main+0xf8>
  120c:       31 c0                   xor    %eax,%eax
  120e:       66 90                   xchg   %ax,%ax
}
```



Analysis:
Compare to the original code with O2, O3, the loop reversal version is a little bit slower than/ almost the same as the original ones in both O2 and O3 optimization.

According to the assembly code, the inline version generate less assembly code than original one, it also avoid function call here. Since it doesn't do loop optimization, so the performance should be similar to original one.

(d)

Among all six methods I applied for this function, only three of them could defeat the compiler with corresponding Optimization level for sure:

Loop unrolling: defeat compiler with O2
Loop reversal: defeat compiler with O2
Loop strip mining: defeat compiler with O3

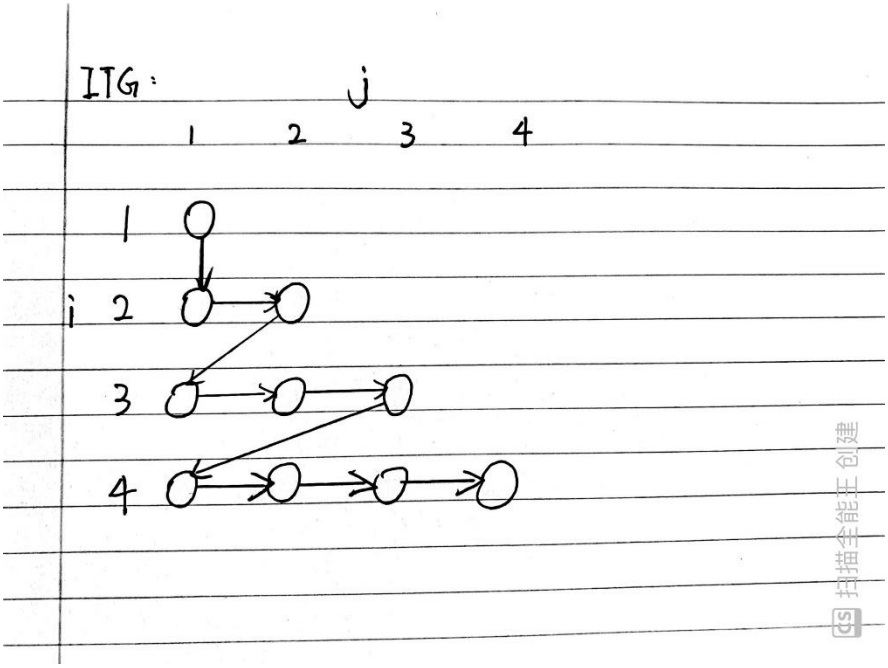
The performance of rest of methods are too close to the original code's, so it is hard to determine whether they defeat compiler for sure.

Question2 :

(a)

We assume N = 4 here

ITG of this code:



(b)

When j=1, i=1. Loop 1
S1: a[1][1] = b[1][1]+c[1][1]*a[2][0];
S2: b[1][1] = a[0][0]*c[0][1];
S3: c[2][1] = a[1][1];
S4: d[1][1] = d[0][2];

When j=2, i=1 Loop 2
S1: a[2][1] = b[2][1]+c[2][1]*a[3][0];
S2: b[2][1] = a[1][0]*c[1][1];
S3: c[3][1] = a[2][1];
S4: d[2][1] = d[1][2];

When j=2, i=2 Loop3
S1: a[2][2] = b[2][2]+c[2][2]*a[3][1];
S2: b[2][2] = a[1][1]*c[1][2];

S3: c[3][2] = a[2][2];
S4: d[2][2] = d[1][3];

When j=3, i=1 Loop4
S1: a[3][1] = b[3][1]+c[3][1]*a[4][0];
S2: b[3][1] = a[2][0]*c[2][1];
S3: c[4][1] = a[3][1];
S4: d[3][1] = d[2][2];

When j=3, i=2 Loop5
S1: a[3][2] = b[3][2]+c[3][2]*a[4][1];
S2: b[3][2] = a[2][1]*c[2][2];
S3: c[4][2] = a[3][2];
S4: d[3][2] = d[2][3];

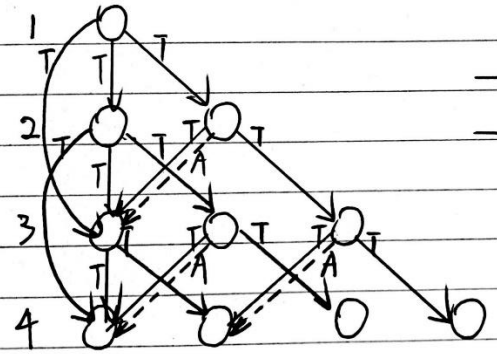
Within the loop, we have loop-independent dependence:
S1->S2 (**anti**): b[i][j]
S1->S3 (**true**): a[i][j]

Between loop, we have loop-carried dependence
Loop1 S1-> Loop3 S2 (**true**): a[i][j] with a[i-1][j-1];
Loop1 S3-> Loop2 S1 (**true**): c[i+1][j] with c[i][j]
Loop1 S3->Loop4 S2(**true**): c[i+1][j] with c[i-1][j]
Loop3 S4->Loop4 S4(**true**): d[i][j] with d[i-1][j+1]
Loop3 S1->Loop4 S1(**anti**): a[i+1][j-1] with a[i][j]

(c)

We still assume N = 4 here

LDG: 1 2 3 4



——: true dependence

——: anti dependence

Question3 :

(a)

Execute time(mili)	Mode for add()		
Cmd line argument	inline	No-inline	original
10,000,000	9.383	18.682	9.628
100,000,000	97.081	184.207	96.58

(b)

For inline version:

```
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>
```

For non_inline version:

```
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>
```

(c)
I expect the inline version runs much faster than no-inline version

According to the screen shoot above, it matches my expectation. Although the no-inline version seems shorter, it call add function by **callq 1840 <_Z3addii>** commend and jump there to execute. When it finish, jumps back. While in inline version, the “add” function is done “in place”, without additional execution jumping, saving a lot of time.

(d)
According to the performance form I record, the running time of original code is very close to the inline version, so compiler should use ”in-lining” by default.

Question4 :

Original code:

```
Int a[N][4];
Int rand_number = rand();
For (i=0; i<4; i++) {
    Threshold = 2.0 * rand_number;
    For (j=0; j<N; j++) {
        If (threshold < 4) {
            Sum = sum + a[j][i];
        } else {
            Sum = sum + a[j][i] + 1;
        }
    }
}
```

Opimization1: Loop Invariant hoisting

```
Int a[N][4];
Int rand_number = rand();
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;
            }
        }
}
```

Opimization2: Loop Unrolling

```
Int a[N][4];
Int rand_number = rand();
For (i=0; i<4; i++) {
        Threshold = 2.0 * rand_number;
        If (threshold < 4) {
                Sum = sum + a[j][i];
                Sum = sum + a[j+1][i];
                …
                Sum = sum + a[j+N-1][i];
            } else {
                Sum = sum + a[j][i] + 1;
                Sum = sum + a[j+1][i] + 1;
                …
                Sum = sum + a[j+N-1][i] + 1;
            }
        }
}
```

Opimization3: Loop Peeling

```
Int a[N][4];
Int rand_number = rand();
For (i=0; i<4; i++) {
        Threshold = 2.0 * rand_number;
        If (threshold < 4) {
                Sum = sum + a[0][i];
            } else {
                Sum = sum + a[0][i] + 1;
            }
        For (j=1; j<N; j++) {
            If (threshold < 4) {
                Sum = sum + a[j][i];
            } else {
                Sum = sum + a[j][i] + 1;
            }
        }
```

```
}
```

Opimization4: Loop Unswitching

```
Int a[N][4];
Int rand_number = rand();
For (i=0; i<4; i++) {
        Threshold = 2.0 * rand_number;
        If (threshold < 4) {
                For (j=0; j<N; j++) {
                        Sum = sum + a[j][i];
                }
         } else {
                For (j=0; j<N; j++) {
                        Sum = sum + a[j][i] + 1;
                }
        }
}
```

Opimization5: Loop Interchange

```
Int a[N][4];
Int rand_number = rand();
For (j=0; i<N; j++) {
        Threshold = 2.0 * rand_number;
        For (j=0; j<4; i++) {
                If (threshold < 4) {
                        Sum = sum + a[j][i];
                } else {
                        Sum = sum + a[j][i] + 1;
                }
        }
}
```

Opimization6: Loop Unroll and Jam

```
Int a[N][4];
Int rand_number = rand();
For (i=0; i<4; i++) {
        Threshold = 2.0 * rand_number;
        For (j=0; j<N; j+=2) {
```

```
            If (threshold < 4) {
                Sum = sum + a[j][i];
                Sum = sum + a[j+1][i];
            } else {
                Sum = sum + a[j][i] + 1;
                Sum = sum + a[j+1][i] + 1;
            }
        }
}
```

Opimization7: Loop Strip and Mining

```
Int a[N][4];
Int rand_number = rand();
For (i=0; i<4; i++) {
        Threshold = 2.0 * rand_number;
        For (j=0; j<N; j+=32) {
            For(int k = j; k < (j+32); k++) {
                If (threshold < 4) {
                    Sum = sum + a[j][i];
                } else {
                    Sum = sum + a[j][i] + 1;
                }
            }
        }
}
```

Question4 :

(a) Loop fusion:

Unsafe

In original code, there exist an anti-dependence between S2 and S3. After the transformation, the dependence changes to loop-carried output dependence between S3 and S1.

(b) Loop Interchange:

Unsafe

In original code, there exist an loop carried anti-dependence between S1, i=I, j=2 and s1, i=2, j=1.(a[1][2] = a[2][1] and a[2][1] = a[3][0]). After the transformation, it changes to loop-carried true dependence between these two instruction (j=1, i=2 and j=2, i=1) (a[2][1] = a[2][1] and a[1][2] = a[2][1]).

(c) Loop Fission:

Safe

In original code, it's not a dependence cycle. There is a loop carried true dependence between i=1 S1 with i=3 S2. After the transformation, it becomes loop-independence true dependence between S1 and S2.