1.Code Optimization – “Beat the Compiler”

(a)

(i) gcc -O2 -o loop_performance loop_performance.c

- *./loop_performance 10000000*

  run 10 times, the shortest time is 23.576000 milliseconds

- *./loop_performance 100000000*

  run 10 times, the shortest time is 232.444000 milliseconds

(ii) gcc -O3 -o loop_performance loop_performance.c

- *./loop_performance 10000000*

  run 10 times, the shortest time is 19.212000 milliseconds

- *./loop_performance 100000000*

  run 10 times, the shortest time is 187.759000 milliseconds

(b)

- I was running the code on a VM provided by duke vcm and the processor arch is x86_64, cpu freq is 2.70GHz, OS is Ubuntu18.

(c)

- Original

```
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;
  }
  for (i=1; i<N; i++) {
    c[i] = b[i-1] + 2;
  }
}
```

- Loop Fusion

By doing so, the optimized -O3 version is slower than original -O3, and the optimized -O2 version is about the same with the original -O2. Therefore, I think the parallellism is kind of hampered in the optimized version.

- Loop unrolling

By doing so, the number of loops is fewer by seeing the assembly code.

- Loop reversal

I reversed the loop order of the loops containing b and c. This did not work well. The -O2 version was about the same while the -O3 version is even slower.

- Final code (Combined method)

```
void do_loops(int *a, int *b, int *c, int N)
{
  int i;
  b[N-1] = a[N] + 3;
  for (i=N-1; i>=2; i--) {
    a[i] = a[i] + 1;
    b[i-1] = a[i] + 3;
    c[i] = b[i-1] + 2;
  }
  c[1] = b[0] + 2;
  a[1] = a[1] + 1;
}
```

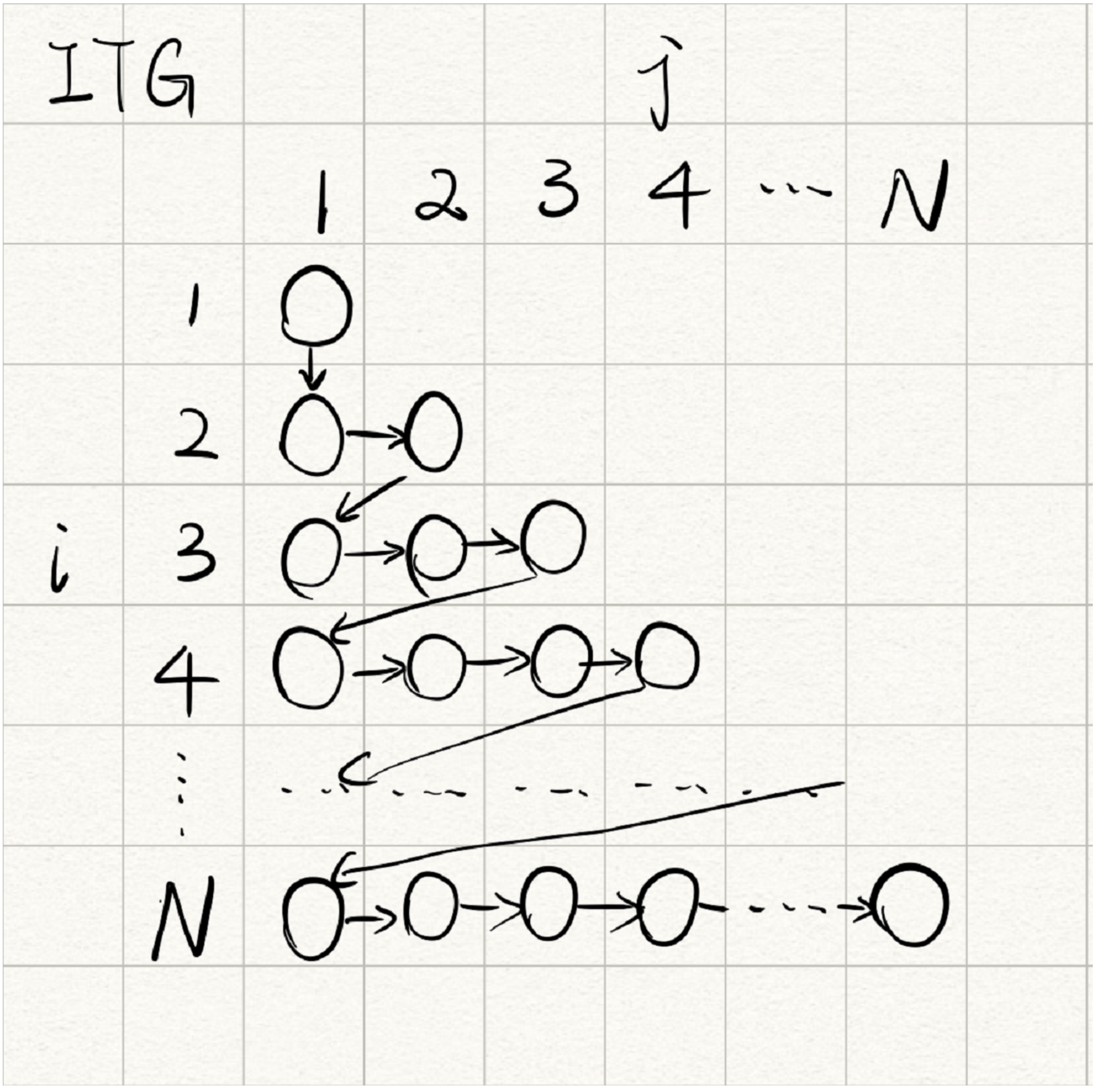
I expected this to work well but it didn't. This could beat the -O2 compiler but not the -O3.

(d)

Generally, I've tried a lot of methods and their combinations, but I cannot beat -O3 compiler. The -O2 compiler is easier to beat. Later I found that this may have something to do with the OS version and gcc version. I'm using Ubuntu18 with gcc 7.5.0, but the same code compiled on a VM with Ubuntu20.04 and gcc 9.1.3 beats the -O3 compiler successfully.

2.Dependence Analysis

(a)



(b)

- Loop independent dependencies:

  S1[i,j] -> T S3[i,j]

  S1[i,j] -> A S2[i,j]

- Loop-carried dependencies:

  S1[i,j] -> T S2[i+1,j+1]

S1[i,j] -> A S1[i+1,j-1]

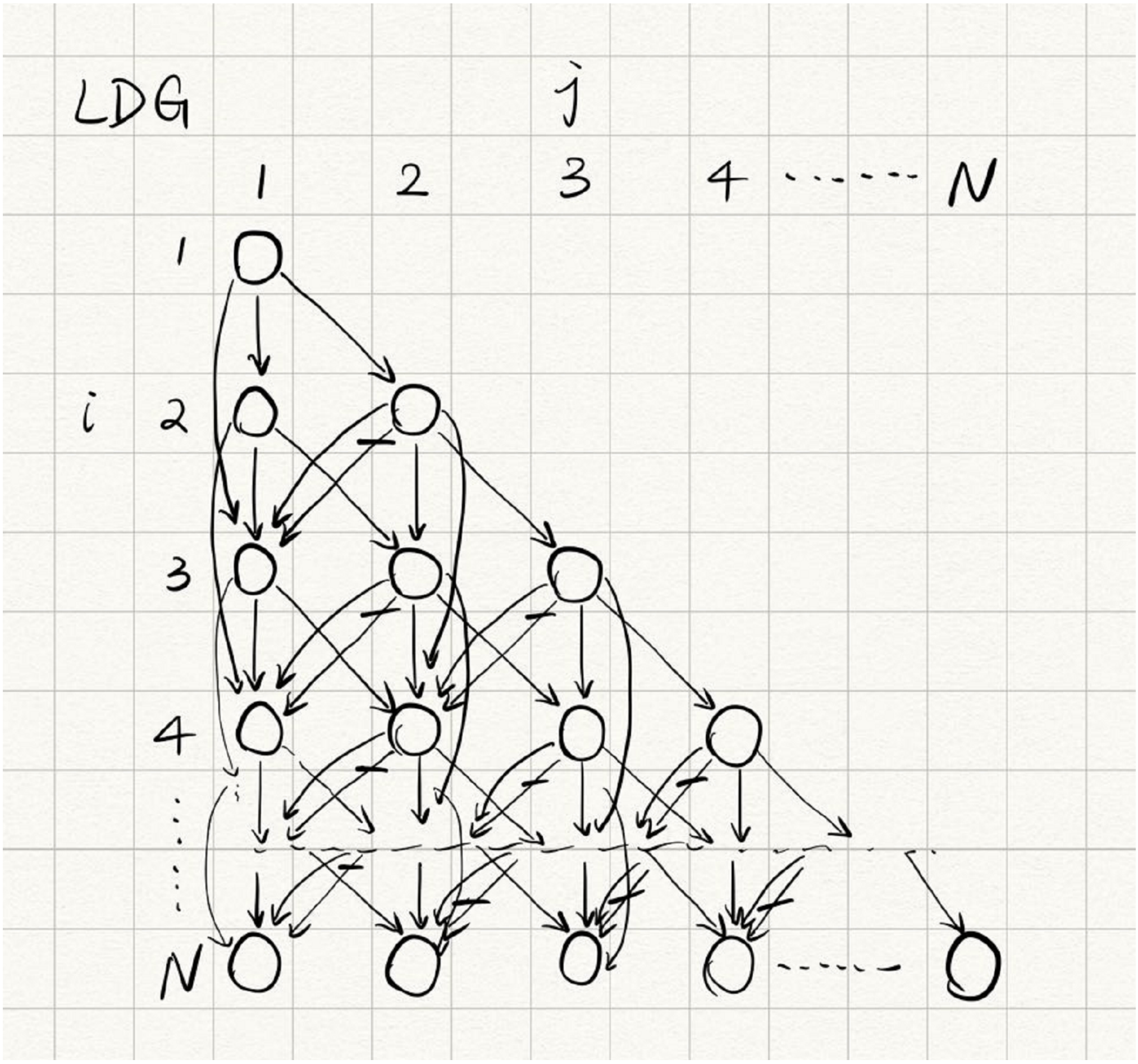
S3[i-1,j] -> T S1[i,j]

S4[i-1,j+1] -> T S4[i,j]

S3[i-2,j] -> T S2[i,j]

(c)

- Loop-carried Dependence Grap



3.Function In-lining and Performance

(a)

(i) with attribute noinline

```
int add (int a, int b) __attribute__ ((__noinline__));

int add  (int a, int b) {
  return (a+b);
}
```

- run 10 times, the shortest time is 254.035 milliseconds

(ii) with attribute always_inline

```
int add (int a, int b) __attribute__ ((__always_inline__));

int add  (int a, int b) {
  return (a+b);
}
```

- run 10 times, the shortest time is 104.934 milliseconds

(b)

noinline assembly code

- loop

```
    e12:    e8 19 fd ff ff          callq  b30 <gettimeofday@plt>
    e17:    44 89 e0                mov    %r12d,%eax
    e1a:    4c 8b 54 24 50          mov    0x50(%rsp),%r10
    e1f:    4c 8b 4c 24 30          mov    0x30(%rsp),%r9
    e24:    4c 8b 44 24 70          mov    0x70(%rsp),%r8
    e29:    48 8d 0c 85 04 00 00    lea    0x4(,%rax,4),%rcx
    e30:    00
    e31:    31 d2                   xor    %edx,%edx
    e33:    0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
    e38:    41 8b 34 12             mov    (%r10,%rdx,1),%esi
    e3c:    41 8b 3c 11             mov    (%r9,%rdx,1),%edi
    e40:    e8 eb 04 00 00          callq  1330 <_Z3addii>
    e45:    41 89 04 10             mov    %eax,(%r8,%rdx,1)
    e49:    48 83 c2 04             add    $0x4,%rdx
    e4d:    48 39 ca                cmp    %rcx,%rdx
    e50:    75 e6                   jne    e38 <main+0x2c8>
    e52:    31 f6                   xor    %esi,%esi
    e54:    48 89 ef                mov    %rbp,%rdi
    e57:    e8 d4 fc ff ff          callq  b30 <gettimeofday@plt>
```

- *add()* function

```
0000000000001330 <_Z3addii>:
    1330:   8d 04 37                lea    (%rdi,%rsi,1),%eax
    1333:   c3                      retq
    1334:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    133b:   00 00 00
    133e:   66 90                   xchg   %ax,%ax
```

- It is clear that for noinline code, every time the code calls *add()* function, in assembly code it actually branch to somewhere else for the code.

always inline assembly code

```
    e12:    e8 19 fd ff ff          callq  b30 <gettimeofday@plt>
    e17:    48 8b 7c 24 50          mov    0x50(%rsp),%rdi
    e1c:    48 8b 74 24 70          mov    0x70(%rsp),%rsi
    e21:    4c 8b 44 24 30          mov    0x30(%rsp),%r8
    e26:    48 8d 47 10             lea    0x10(%rdi),%rax
    e2a:    48 8d 4e 10             lea    0x10(%rsi),%rcx
    e2e:    48 39 c6                cmp    %rax,%rsi
    e31:    0f 93 c2                setae  %dl
    e34:    48 39 cf                cmp    %rcx,%rdi
    e37:    0f 93 c0                setae  %al
    e3a:    09 c2                   or     %eax,%edx
    e3c:    49 8d 40 10             lea    0x10(%r8),%rax
    e40:    48 39 c6                cmp    %rax,%rsi
    e43:    0f 93 c0                setae  %al
    e46:    49 39 c8                cmp    %rcx,%r8
    e49:    0f 93 c1                setae  %cl
    e4c:    09 c8                   or     %ecx,%eax
    e4e:    84 c2                   test   %al,%dl
    e50:    0f 84 7e 03 00 00       je     11d4 <main+0x664>
    e56:    83 fb 08                cmp    $0x8,%ebx
    e59:    0f 86 75 03 00 00       jbe    11d4 <main+0x664>
    e5f:    48 89 f9                mov    %rdi,%rcx
```

```
e62:    48 c1 e9 02             shr    $0x2,%rcx
e66:    48 f7 d9                neg    %rcx
e69:    83 e1 03                and    $0x3,%ecx
e6c:    0f 84 5a 03 00 00       je     11cc <main+0x65c>
e72:    41 8b 00                mov    (%r8),%eax
e75:    03 07                   add    (%rdi),%eax
e77:    83 f9 01                cmp    $0x1,%ecx
e7a:    89 06                   mov    %eax,(%rsi)
e7c:    0f 84 b0 03 00 00       je     1232 <main+0x6c2>
e82:    41 8b 40 04             mov    0x4(%r8),%eax
e86:    03 47 04                add    0x4(%rdi),%eax
e89:    83 f9 02                cmp    $0x2,%ecx
e8c:    89 46 04                mov    %eax,0x4(%rsi)
e8f:    0f 84 dc 03 00 00       je     1271 <main+0x701>
e95:    41 8b 40 08             mov    0x8(%r8),%eax
e99:    03 47 08                add    0x8(%rdi),%eax
e9c:    41 b9 03 00 00 00       mov    $0x3,%r9d
ea2:    89 46 08                mov    %eax,0x8(%rsi)
ea5:    41 89 db                mov    %ebx,%r11d
ea8:    31 c0                   xor    %eax,%eax
eaa:    31 d2                   xor    %edx,%edx
eac:    41 29 cb                sub    %ecx,%r11d
eaf:    89 c9                   mov    %ecx,%ecx
eb1:    48 c1 e1 02             shl    $0x2,%rcx
eb5:    45 89 da                mov    %r11d,%r10d
eb8:    4c 8d 34 0f             lea    (%rdi,%rcx,1),%r14
ebc:    4d 8d 2c 08             lea    (%r8,%rcx,1),%r13
ec0:    41 c1 ea 02             shr    $0x2,%r10d
ec4:    48 01 f1                add    %rsi,%rcx
ec7:    66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
ece:    00 00
ed0:    f3 41 0f 6f 44 05 00    movdqu 0x0(%r13,%rax,1),%xmm0
ed7:    83 c2 01                add    $0x1,%edx
eda:    66 41 0f fe 04 06       paddd  (%r14,%rax,1),%xmm0
ee0:    0f 11 04 01             movups %xmm0,(%rcx,%rax,1)
ee4:    48 83 c0 10             add    $0x10,%rax
ee8:    41 39 d2                cmp    %edx,%r10d
eeb:    77 e3                   ja     ed0 <main+0x360>
eed:    44 89 da                mov    %r11d,%edx
ef0:    83 e2 fc                and    $0xfffffffc,%edx
ef3:    41 39 d3                cmp    %edx,%r11d
ef6:    42 8d 04 0a             lea    (%rdx,%r9,1),%eax
efa:    74 70                   je     f6c <main+0x3fc>
efc:    48 63 d0                movslq %eax,%rdx
eff:    41 8b 0c 90             mov    (%r8,%rdx,4),%ecx
f03:    03 0c 97                add    (%rdi,%rdx,4),%ecx
f06:    89 0c 96                mov    %ecx,(%rsi,%rdx,4)
f09:    8d 50 01                lea    0x1(%rax),%edx
f0c:    39 d3                   cmp    %edx,%ebx
f0e:    7e 5c                   jle    f6c <main+0x3fc>
f10:    48 63 d2                movslq %edx,%rdx
f13:    41 8b 0c 90             mov    (%r8,%rdx,4),%ecx
f17:    03 0c 97                add    (%rdi,%rdx,4),%ecx
f1a:    89 0c 96                mov    %ecx,(%rsi,%rdx,4)
f1d:    8d 50 02                lea    0x2(%rax),%edx
f20:    39 d3                   cmp    %edx,%ebx
f22:    7e 48                   jle    f6c <main+0x3fc>
f24:    48 63 d2                movslq %edx,%rdx
f27:    41 8b 0c 90             mov    (%r8,%rdx,4),%ecx
f2b:    03 0c 97                add    (%rdi,%rdx,4),%ecx
f2e:    89 0c 96                mov    %ecx,(%rsi,%rdx,4)
f31:    8d 50 03                lea    0x3(%rax),%edx
f34:    39 d3                   cmp    %edx,%ebx
f36:    7e 34                   jle    f6c <main+0x3fc>
f38:    48 63 d2                movslq %edx,%rdx
f3b:    41 8b 0c 90             mov    (%r8,%rdx,4),%ecx
f3f:    03 0c 97                add    (%rdi,%rdx,4),%ecx
f42:    89 0c 96                mov    %ecx,(%rsi,%rdx,4)
f45:    8d 50 04                lea    0x4(%rax),%edx
f48:    39 d3                   cmp    %edx,%ebx
f4a:    7e 20                   jle    f6c <main+0x3fc>
f4c:    48 63 d2                movslq %edx,%rdx
f4f:    83 c0 05                add    $0x5,%eax
f52:    41 8b 0c 90             mov    (%r8,%rdx,4),%ecx
f56:    03 0c 97                add    (%rdi,%rdx,4),%ecx
f59:    39 c3                   cmp    %eax,%ebx
f5b:    89 0c 96                mov    %ecx,(%rsi,%rdx,4)
f5e:    7e 0c                   jle    f6c <main+0x3fc>
f60:    48 98                   cltq
```

```
f62:    41 8b 14 80             mov    (%r8,%rax,4),%edx
f66:    03 14 87                add    (%rdi,%rax,4),%edx
f69:    89 14 86                mov    %edx,(%rsi,%rax,4)
f6c:    31 f6                   xor    %esi,%esi
f6e:    48 89 ef                mov    %rbp,%rdi
f71:    e8 ba fb ff ff          callq  b30 <gettimeofday@plt>
```

- While in always_inline code, the loop was flattened and execute many times, without branching soemwhere else, which significantly reduced the instruction number executed.

(c)

- Yes, it matched my expectation. The time cost of code compiled inline should be less than the one compiled noinline, since the total instructions executed was less.

(d)

- For no attribute on *add()* function, out of 10 times, the shortest time is 104.594 milliseconds. I would give a guess that the compiler was in-lining the *add()* function by default.

4.Loop transformations I

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;
        }
    }
}
```

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;
        }
    }
}
```

Loop unswitching

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

Loop Interchange

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

Loop Unrolling

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

5.Loop transformations II

(a)

- Not safe. Originally, there is a loop-carried output dependency from S1[i] to S3[i-1] and a loop-carried anti dependency from S2[i] to S3[i-1], while after the transformation, there is a loop-carried output dependency from S3[i] to S1[i+1] and a loop-carried true dependency from S3[i] to S2[i+1].

(b)

- Not safe. Originally, the outermost loop carries a loop-carried anti dependency from S1[i][j] to S1[i+1][j-1]. Here i < i' and j > j', so it's not safe.

(c)

- Safe.