Rijish Ganguly
rg239
ECE 565

1a)

Unmodified source code:

a. Optimization – O3
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	20.26	20.13	20.47	19.80	24.19	20.97	19.80
100,000,000	207.12	200.54	199.86	200.88	201.48	201.97	199.86

The shortest run time **for O3 with N = 10,000,000 is 19.80**
The average run time **for O3 with N = 10,000,000 is 20.97**

The shortest run time **for O3 with N = 100,000,000 is 199.86**
The average run time **for O3 with N = 100,000,000 is 201**

b. Optimization – O2
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	19.61	20.45	20.37	23.27	20.20	20.78	19.61
100,000,000	199.00	198.55	198.77	210.63	204.78	202.34	198.55

The shortest run time **for O2 with N = 10,000,000 is 19.61**
The average run time **for O3 with N = 10,000,000 is 20.78**

The shortest run time **for O3 with N = 100,000,000 is 198.55**
The average run time **for O3 with N = 100,000,000 is 202.34**

1b)
Processor Architecture – x86 (64-bits)
CPU Frequency: 1.8 GHz
OS type: macOS Mojave
Standalone system

1c)

Optimization I

Loop Fusion

Resulting Code:

```
Loop Fusion

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;
  }
}
```

Optimization – O3

Output sum: 69999988

Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	23.66	24.31	24.18	24.53	26.08	24.55	23.66
100,000,000	586.62	268.78	250.74	253.58	250.05	321.95	250.05

Optimization – O2

Output sum: 69999988

Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	32.75	43.28	33.38	25.91	39.06	34.876	25.91
100,000,000	260.04	255.84	255.38	249.36	253.94	254.91	249.36

The optimization didn’t improve the performance of our code. The performance worsened compared to the original unmodified source code. Loop fusion may not improve performance as two separate loops might perform better on an architecture because of data locality.

Optimization II
Loop Peeling
Resulting Code:

```
Loop Peeling

void do_loops(int *a, int *b, int *c, int N)
{
  int i;

  for (i=N-1; i>=1; i-=1) {
    a[i] = a[i] + 1;
  }

  if(N > 0){
    b[1] = a[2] + 3;
  }

  for (i=2; i<N; i++) {
    b[i] = a[i+1] + 3;
  }

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

Optimization – O3
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	28.28	29.20	21.35	21.36	27.41	25.52	21.35
100,000,000	242.08	202.04	201.59	201.11	201.97	209.75	201.11

Optimization – O2
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	21.10	29.86	21.10	20.90	22.13	23.01	20.90
100,000,000	223.03	201.55	200.97	202.64	200.11	205.66	200.11

The optimization didn’t improve the performance of our code. The performance of the modified code is comparable to the unmodified source code. However, the performance of the unmodified source code is better by a few milliseconds.

Optimization III
Loop Reversal
Resulting Code:

```
Loop Reversal

void do_loops(int *a, int *b, int *c, int N)
{
  int i;

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

Optimization – O3
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	21.04	22.98	27.79	20.97	33.52	25.26	21.04
100,000,000	205.24	195.76	195.95	198.53	199.13	198.92	195.76

Optimization – O2
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	21.26	22.22	20.69	23.25	20.94	21.67	20.69
100,000,000	230.76	200.53	200.05	203.38	232.41	213.42	200.05

The performance was improved for the modified code. For optimization level O3 and N =100,000,000, the shortest run time was 195.76 ms which was faster than the shortest run time for the original code which was 199.86 ms. The average run time was also improved for this case. For optimization level O3 and N = 10,0000, the performance of the modified and the unmodified code was comparable. For optimization level O2, the performance was almost similar to the unmodified code. Loop reversal can be beneficial because some ISAs contain efficient loop count instructions that count in a single direction.

Optimization IV
Loop Strip Mining
Resulting Code:

```
Loop Strip Mining

void do_loops(int *a, int *b, int *c, int N)
{
  int i;
  int j;

  for (i=N-1; i>=1; i--) {
    a[i] = a[i] + 1;
  }

  for (j=1; j<N; j+=100) {
    for (i=j; i < j+100; i++){
       b[i] = a[i+1] + 3;
  }
  }

  for(j=1; j<N; j+=100){
  for (i=j; i< j+100; i++) {
    c[i] = b[i-1] + 2;
  }
}
}
```

Optimization – O3
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	21.33	26.61	21.17	21.23	21.28	22.32	21.23
100,000,000	619.60	255.99	272.17	245.81	235.14	325.74	235.14

Optimization – O2
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	21.49	29.31	21.17	21.47	27.47	24.18	21.17
100,000,000	203.97	210.91	201.46	201.32	201.92	203.91	201.32

The optimization didn't improve the performance of our code. The performance of the modified code was significantly worse for O3 optimization with N = 100,000,000. The performance for O2 optimization was comparable to the unmodified code. Vectorization of the loop didn't yield significant benefit because of processor characteristics.

Optimization V
Loop Unrolling
Resulting Code:

```
Loop Unrolling

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+=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;
  }
}
```

Optimization – O3
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	30.15	26.45	29.52	32.34	36.93	31.07	26.45
100,000,000	233.25	213.10	207.22	210.03	207.73	214.26	207.22

Optimization – O2
Output sum: 69999988
Runtime is in milliseconds

N	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Shortest
10,000,000	31.32	29.19	28.56	26.45	32.03	29.51	26.45

100,000,000	1148.70	392.75	304.56	215.83	537.39	519.84	215.83

The optimization didn't improve the performance of our code. The performance was significantly worse for the case of O2 optimization with N = 100,000,000. For, all the other cases, the unmodified code performed better than the modified code.

Assembly Code analysis:

a. Unmodified Code:

```
0000000100000c06        subl    $0x1, %ecx
0000000100000c09        movl    %ecx, -0x20(%rbp)
0000000100000c0c        cmpl    $0x1, -0x20(%rbp)
0000000100000c10        jl      0x100000c3d
0000000100000c16        movq    -0x8(%rbp), %rax
0000000100000c1a        movslq  -0x20(%rbp), %rcx
0000000100000c1e        movl    (%rax,%rcx,4), %edx
0000000100000c21        addl    $0x1, %edx
0000000100000c24        movq    -0x8(%rbp), %rax
0000000100000c28        movslq  -0x20(%rbp), %rcx
0000000100000c2c        movl    %edx, (%rax,%rcx,4)
0000000100000c2f        movl    -0x20(%rbp), %eax
0000000100000c32        addl    $-0x1, %eax
0000000100000c35        movl    %eax, -0x20(%rbp)
0000000100000c38        jmp     0x100000c0c
0000000100000c3d        movl    $0x1, -0x20(%rbp)
0000000100000c44        movl    -0x20(%rbp), %eax
0000000100000c47        cmpl    -0x1c(%rbp), %eax
0000000100000c4a        jge     0x100000c7c
0000000100000c50        movq    -0x8(%rbp), %rax
0000000100000c54        movl    -0x20(%rbp), %ecx
0000000100000c57        addl    $0x1, %ecx
0000000100000c5a        movslq  %ecx, %rdx
0000000100000c5d        movl    (%rax,%rdx,4), %ecx
0000000100000c60        addl    $0x3, %ecx
0000000100000c63        movq    -0x10(%rbp), %rax
0000000100000c67        movslq  -0x20(%rbp), %rdx
0000000100000c6b        movl    %ecx, (%rax,%rdx,4)
0000000100000c6e        movl    -0x20(%rbp), %eax
0000000100000c71        addl    $0x1, %eax
0000000100000c74        movl    %eax, -0x20(%rbp)
0000000100000c77        jmp     0x100000c44
0000000100000c7c        movl    $0x1, -0x20(%rbp)
0000000100000c83        movl    -0x20(%rbp), %eax
0000000100000c86        cmpl    -0x1c(%rbp), %eax
0000000100000c89        jge     0x100000cbb
0000000100000c8f        movq    -0x10(%rbp), %rax
```

In the unmodified code, we observe that three loops are being executed with the help of jump and compare statements

b. <u>Loop Fusion</u>:

```
_do_loops:
0000000100000c10        pushq   %rbp
0000000100000c11        movq    %rsp, %rbp
0000000100000c14        movq    %rdi, -0x8(%rbp)
0000000100000c18        movq    %rsi, -0x10(%rbp)
0000000100000c1c        movq    %rdx, -0x18(%rbp)
0000000100000c20        movl    %ecx, -0x1c(%rbp)
0000000100000c23        movl    -0x1c(%rbp), %ecx
0000000100000c26        subl    $0x1, %ecx
0000000100000c29        movl    %ecx, -0x20(%rbp)
0000000100000c2c        cmpl    $0x1, -0x20(%rbp)
0000000100000c30        jl      0x100000c5d
0000000100000c36        movq    -0x8(%rbp), %rax
0000000100000c3a        movslq  -0x20(%rbp), %rcx
0000000100000c3e        movl    (%rax,%rcx,4), %edx
0000000100000c41        addl    $0x1, %edx
0000000100000c44        movq    -0x8(%rbp), %rax
0000000100000c48        movslq  -0x20(%rbp), %rcx
0000000100000c4c        movl    %edx, (%rax,%rcx,4)
0000000100000c4f        movl    -0x20(%rbp), %eax
0000000100000c52        addl    $-0x1, %eax
0000000100000c55        movl    %eax, -0x20(%rbp)
0000000100000c58        jmp     0x100000c2c
0000000100000c5d        movl    $0x1, -0x20(%rbp)
0000000100000c64        movl    -0x20(%rbp), %eax
0000000100000c67        cmpl    -0x1c(%rbp), %eax
0000000100000c6a        jge     0x100000cba
0000000100000c70        movq    -0x8(%rbp), %rax
0000000100000c74        movl    -0x20(%rbp), %ecx
0000000100000c77        addl    $0x1, %ecx
0000000100000c7a        movslq  %ecx, %rdx
0000000100000c7d        movl    (%rax,%rdx,4), %ecx
0000000100000c80        addl    $0x3, %ecx
0000000100000c83        movq    -0x10(%rbp), %rax
0000000100000c87        movslq  -0x20(%rbp), %rdx
0000000100000c8b        movl    %ecx, (%rax,%rdx,4)
0000000100000c8e        movq    -0x10(%rbp), %rax
```

In the loop-fusion modification, we observe that two loops are being executed with the help of jump and compare statements.

c. Loop Peeling:

```
0000000100000c07        movq    -0x8(%rbp), %rax
0000000100000c0b        movl    0x8(%rax), %ecx
0000000100000c0e        addl    $0x3, %ecx
0000000100000c11        movq    -0x10(%rbp), %rax
0000000100000c15        movl    %ecx, 0x4(%rax)
0000000100000c18        movl    $0x2, -0x20(%rbp)
0000000100000c1f        movl    -0x20(%rbp), %eax
0000000100000c22        cmpl    -0x1c(%rbp), %eax
0000000100000c25        jge     0x100000c57
0000000100000c2b        movq    -0x8(%rbp), %rax
0000000100000c2f        movl    -0x20(%rbp), %ecx
0000000100000c32        addl    $0x1, %ecx
0000000100000c35        movslq  %ecx, %rdx
0000000100000c38        movl    (%rax,%rdx,4), %ecx
0000000100000c3b        addl    $0x3, %ecx
0000000100000c3e        movq    -0x10(%rbp), %rax
0000000100000c42        movslq  -0x20(%rbp), %rdx
0000000100000c46        movl    %ecx, (%rax,%rdx,4)
```

For the loop peeling example, we observe the addition of 3 to the first element of the arrays, before the loop initiates.

d. Loop Reversal:

```
0000000100000bfa        movl    -0x20(%rbp), %eax
0000000100000bfd        movl    -0x1c(%rbp), %ecx
0000000100000c00        subl    $0x1, %ecx
0000000100000c03        cmpl    %ecx, %eax
0000000100000c05        jg      0x100000c32
0000000100000c0b        movq    -0x8(%rbp), %rax
0000000100000c0f        movslq  -0x20(%rbp), %rcx
0000000100000c13        movl    (%rax,%rcx,4), %edx
0000000100000c16        addl    $0x1, %edx
0000000100000c19        movq    -0x8(%rbp), %rax
0000000100000c1d        movslq  -0x20(%rbp), %rcx
0000000100000c21        movl    %edx, (%rax,%rcx,4)
0000000100000c24        movl    -0x20(%rbp), %eax
0000000100000c27        addl    $0x1, %eax
0000000100000c2a        movl    %eax, -0x20(%rbp)
0000000100000c2d        jmp     0x100000bfa
```

For loop reversal, we observe that the code is similar to the unmodified code, however, the order of loop has been reversed and the loop counter is decremented with the help of the subl instruction.

e.

<u>Loop Strip-Mining</u>:

```
0000000100000bcc        movl     %edx, (%rax,%rcx,4)
0000000100000bcf        movl     -0x20(%rbp), %eax
0000000100000bd2        addl     $-0x1, %eax
0000000100000bd5        movl     %eax, -0x20(%rbp)
0000000100000bd8        jmp      0x100000bac
0000000100000bdd        movl     $0x1, -0x24(%rbp)
0000000100000be4        movl     -0x24(%rbp), %eax
0000000100000be7        cmpl     -0x1c(%rbp), %eax
0000000100000bea        jge      0x100000c46
0000000100000bf0        movl     -0x24(%rbp), %eax
0000000100000bf3        movl     %eax, -0x20(%rbp)
0000000100000bf6        movl     -0x20(%rbp), %eax
0000000100000bf9        movl     -0x24(%rbp), %ecx
0000000100000bfc        addl     $0x64, %ecx
0000000100000bff        cmpl     %ecx, %eax
0000000100000c01        jge      0x100000c33
0000000100000c07        movq     -0x8(%rbp), %rax
0000000100000c0b        movl     -0x20(%rbp), %ecx
0000000100000c0e        addl     $0x1, %ecx
0000000100000c11        movslq   %ecx, %rdx
0000000100000c14        movl     (%rax,%rdx,4), %ecx
0000000100000c17        addl     $0x3, %ecx
0000000100000c1a        movq     -0x10(%rbp), %rax
0000000100000c1e        movslq   -0x20(%rbp), %rdx
0000000100000c22        movl     %ecx, (%rax,%rdx,4)
0000000100000c25        movl     -0x20(%rbp), %eax
```

For loop strip-mining, we observe a double nested loop structure. The number of instructions also increased significantly.

f. <u>Partial Loop Unrolling:</u>

```
0000000100000b90        addl    $0x3, %ecx
0000000100000b93        movq    -0x10(%rbp), %rax
0000000100000b97        movslq  -0x20(%rbp), %rdx
0000000100000b9b        movl    %ecx, (%rax,%rdx,4)
0000000100000b9e        movq    -0x8(%rbp), %rax
0000000100000ba2        movl    -0x20(%rbp), %ecx
0000000100000ba5        addl    $0x2, %ecx
0000000100000ba8        movslq  %ecx, %rdx
0000000100000bab        movl    (%rax,%rdx,4), %ecx
0000000100000bae        addl    $0x3, %ecx
0000000100000bb1        movq    -0x10(%rbp), %rax
0000000100000bb5        movl    -0x20(%rbp), %esi
0000000100000bb8        addl    $0x1, %esi
0000000100000bbb        movslq  %esi, %rdx
0000000100000bbe        movl    %ecx, (%rax,%rdx,4)
0000000100000bc1        movq    -0x8(%rbp), %rax
0000000100000bc5        movl    -0x20(%rbp), %ecx
0000000100000bc8        addl    $0x3, %ecx
0000000100000bcb        movslq  %ecx, %rdx
0000000100000bce        movl    (%rax,%rdx,4), %ecx
0000000100000bd1        addl    $0x3, %ecx
0000000100000bd4        movq    -0x10(%rbp), %rax
0000000100000bd8        movl    -0x20(%rbp), %esi
0000000100000bdb        addl    $0x2, %esi
0000000100000bde        movslq  %esi, %rdx
0000000100000be1        movl    %ecx, (%rax,%rdx,4)
0000000100000be4        movq    -0x8(%rbp), %rax
0000000100000be8        movl    -0x20(%rbp), %ecx
0000000100000beb        addl    $0x4, %ecx
```

For the partial loop unrolling, we observe more statements being executed within the loop and the number of instructions increase significantly.

1d)

I could beat the performance of the compiler for only case when I optimized the loop with the help of loop reversal. On an average, the performance of the compiler was significantly better for other scenarios. Compilers are great at knowing the better way to perform an operation or sequence of operations in the context of the target and compilation objectives. Hence, the overall optimization done by the compiler produced better results than my standalone modifications.

5. Loop Transformations

(a) Loop Fusion

Original Dependencies:
S1 => T S2 (loop independent)
S1 => O S3 (loop carried)
S2 => A S3 (loop carried)

After Loop Fusion, the dependencies become as follow:
S1 => T S2(loop independent)

S1 => O S3(loop carried)
S2 => T S3(loop carried)

Loop fusion is safe if and only if no data dependence between the nests becomes loop-carried data dependence of a different type. We observe that the condition is violated for the above loop fusion.

(b) Loop Interchange

Original Dependencies:
S1 => A S1 (loop dependent)
We observe that there is a data dependence carried by the outer loop executed for i and j to another statement instance i' and j' where i < i' and j > j'.
Example: (1,3) => (2,2)

Loop Interchange is safe if outermost loop does not carry any data dependence from one statement instance executed for i and j to another statement instance executed for i' and j' where (i < i' and j > j') OR (i > i' and j < j'). Since this condition is violated, loop interchange won't be safe

(c) Loop Fission

Original Dependencies:
S1 => T S2(loop dependent)

After loop fission:
S1 => T S2(loop dependent)

Loop Fission is safe if and only if statements involved in a cycle of loop-carried data dependences remain in the same loop and if there exists a data dependence between two statements placed in different loops, the dependence type must not change. There is no violation of condition for this case. Hence, it is safe to use loop fission.

6. Loop Transformation

Initial 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;
     }
  }
}
```

A. <u>Loop Invariant</u>

Pulling non-loop-dependent calculations out of the loop

```
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;
     }
  }
}
```

B. Loop un-switching

Move a conditional expression outside of a loop, and replicate loop body inside of each conditional block

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

C. Loop Interchange

Switch the positions of one loop that is tightly nested within another loop

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

D. Loop Peeling

Remove first and/or last iterations of a loop body to separate code outside the loop

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

E. Loop Unrolling

Combine multiple instances of the loop body and make corresponding reduction to the loop iteration count

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

F. <u>Loop Reversal</u>

Reverse the order of the loop iteration

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

3. Function in-lining and performance

(a)

In-lining the add function

```
checksum=-2120047872
Time=101.488 milliseconds
checksum=-2120047872
Time=101.525 milliseconds
checksum=-2120047872
Time=121.764 milliseconds
checksum=-2120047872
Time=104.033 milliseconds
checksum=-2120047872
Time=102.093 milliseconds
```

Average Time: 106.1806 ms
Shortest Run-time: 101.488 ms

Never In-lining the add function

```
checksum=-2120047872
Time=228.806 milliseconds
checksum=-2120047872
Time=211.334 milliseconds
checksum=-2120047872
Time=202.758 milliseconds
checksum=-2120047872
Time=198.533 milliseconds
checksum=-2120047872
Time=202.362 milliseconds
```

Average Time: 208.7586
Shortest Run-time: 202.362 ms

(b) Effects of in-lining:

Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

Code-snippets for in-lining:

```
00000001000014fd        callq   0x100001cc0 ## symbol stub for: _gettimeofday
0000000100001502        movq    -0x38(%rbp), %rsi
0000000100001506        movl    %r12d, -0x50(%rbp)
000000010000150a        cmpl    $0x7, %r12d
000000010000150e        jbe     0x100001600
```

```
0000000100001600        movl    (%r15,%rbx,4), %eax
0000000100001604        addl    (%r14,%rbx,4), %eax
0000000100001608        movl    %eax, (%r13,%rbx,4)
000000010000160d        incq    %rbx
0000000100001610        cmpq    %rbx, %rsi
0000000100001613        jne     0x100001600
0000000100001615        xorl    %r12d, %r12d
0000000100001618        leaq    -0x70(%rbp), %rdi
000000010000161c        xorl    %esi, %esi
000000010000161e        callq   0x100001cc0 ## symbol stub for: _gettimeofday
```

The loop being carried out

```
__Z3addii:
0000000100001290        pushq   %rbp
0000000100001291        movq    %rsp, %rbp
0000000100001294        leal    (%rdi,%rsi), %eax
0000000100001297        popq    %rbp
0000000100001298        retq
0000000100001299        nopl    (%rax)
```

We don't observe any explicit call to the addition function (_Z3addii) as it gets expanded within another function

Code snippet-for no in-lining:

```
0000000100001609        callq   0x100001cc0 ## symbol stub for: _gettimeofday
000000010000160e        nop
0000000100001610        movl    (%r12,%rbx,4), %edi
0000000100001614        movl    (%r15,%rbx,4), %esi
0000000100001618        callq   __Z3addii ## add(int, int)
000000010000161d        movl    %eax, (%r13,%rbx,4)
0000000100001622        incq    %rbx
0000000100001625        cmpq    %rbx, %r14
0000000100001628        jne     0x100001610
000000010000162a        xorl    %r12d, %r12d
000000010000162d        leaq    -0x78(%rbp), %rdi
0000000100001631        xorl    %esi, %esi
0000000100001633        callq   0x100001cc0 ## symbol stub for: _gettimeofday
```

We observe an explicit call to the addition function

(c) My measured performance results match my expectations. The average runtime of the program with always in-lining attribute was almost approximately half of the average runtime of the program with no in-lining. In-lining reduces the number of executed instructions by avoiding function calls and return instructions whereas context-switch can prove to be costly for small functions. For our program we are calling the addition function several times and since the size of the function is relatively small, we get superior performance with in-lining.

(d) Performance of original code:

```
checksum=-2120047872
Time=101.374 milliseconds
checksum=-2120047872
Time=101.923 milliseconds
checksum=-2120047872
Time=106.374 milliseconds
checksum=-2120047872
Time=108.155 milliseconds
checksum=-2120047872
Time=100.814 milliseconds
```

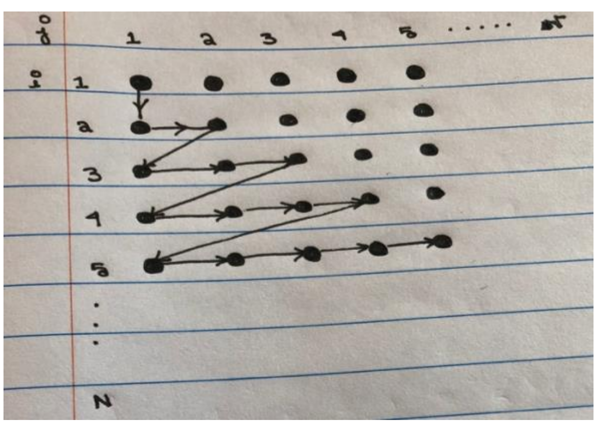
Average Run-time: 103.728 ms
Shortest Run-time: 101.374 ms

The performance of the original code is almost the same as the performance of the in-lined code. Based on this, we can make the claim that the compiler is in-lining the add function.

2.

(a)

<u>Iteration-space Traversal Graph</u>



Iteration: (1,1) => (2,1) => (2,2) => (3,1) => (3,2)

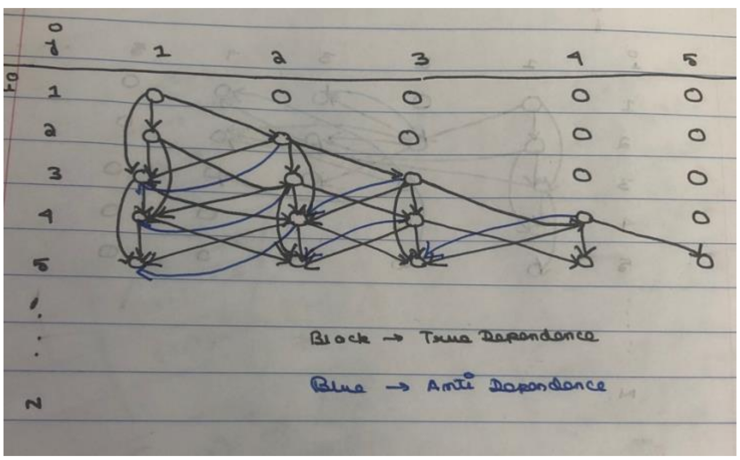
(b)

Dependence:

1. S1 => T S3 — Loop Independent
2. S1 => A S2 — Loop Independent
3. S3[i,j] => T S1[i+1,j] — Loop Carried
4. S4[i,j] => T S4[i+1, j-1] — Loop Carried
5. S1[i,j] => T S2[i+1,j+1] — Loop Carried
6. S1[i,j] => A S1[i+1,j-1] — Loop Carried
7. S3[i,j] => T S2[i+2,j] — Loop Carried

(c)

Loop Dependence Traversal Graph



`