HW1

Yuanrui Zhang (yz545)

1 Code Optimization – "Beat the Compiler"

(a)

Optimization \ Argv	10000000	100000000
-O2	14.500000ms	154.252000ms
-O3	16.582000ms	169.610000ms

*The recorded time is the shortest time seen in a test of 8 each (same in the following tables).

(b)

(1) Processor architecture: Intel Core i7(64-bit x86-64)
(2) CPU frequency: 2.6GHz
(3) OS: MacOS
(4) A standalone system

(c) Note that the disassembly code in the following section is generated with O2 compiler optimization flag.

(1) Loop Unrolling

A. Performance observations

Optimization \ Argv	10000000	100000000
-O2	14.069000ms	154.219000ms
-O3	14.559000ms	154.152000ms

B. Result analysis

It matches my expectation. Loop unrolling does improve performance, especially for O3 optimization tag.

Reason: loop unrolling can add additional instruction level parallelism and reduce the amount of loop management instructions. Compilers with O2 tag are not smart enough to optimize with loop unrolling, however, O3 tag indicates compiler optimize loop unrolling so that an obvious performance enhancement is shown.

C. objdump snapshot

According to the assembly code below, the gray area indicates an unrolling operation. As we can tell, it repeats calculate 9 times, matching the hand-tuned code I did. No doubt, the loop management overhead instruction is reduced.

```
100000ab5:      48 83 c0 f7     addq    $-9, %rax
100000ab7:      7f d7           jg      -41 <_do_loops+0x20>
100000ab9:      83 f9 02        cmpl    $2, %ecx
100000abc:      0f 8c 0f 01 00 00       jl      271 <_do_loops+0x161>
100000ac2:      4c 63 c1        movslq  %ecx, %r8
100000ac5:      41 b9 09 00 00 00       movl    $9, %r9d
100000acb:      0f 1f 44 00 00  nopl    (%rax,%rax)
100000ad0:      42 8b 44 8f e4  movl    -28(%rdi,%r9,4), %eax
100000ad5:      83 c0 03        addl    $3, %eax
100000ad8:      42 89 44 8e e0  movl    %eax, -32(%rsi,%r9,4)
100000add:      42 8b 44 8f e8  movl    -24(%rdi,%r9,4), %eax
100000ae2:      83 c0 03        addl    $3, %eax
100000ae5:      42 89 44 8e e4  movl    %eax, -28(%rsi,%r9,4)
100000aea:      42 8b 44 8f ec  movl    -20(%rdi,%r9,4), %eax
100000aef:      83 c0 03        addl    $3, %eax
100000af2:      42 89 44 8e e8  movl    %eax, -24(%rsi,%r9,4)
100000af7:      42 8b 44 8f f0  movl    -16(%rdi,%r9,4), %eax
100000afc:      83 c0 03        addl    $3, %eax
100000aff:      42 89 44 8e ec  movl    %eax, -20(%rsi,%r9,4)
100000b04:      42 8b 44 8f f4  movl    -12(%rdi,%r9,4), %eax
100000b09:      83 c0 03        addl    $3, %eax
100000b0c:      42 89 44 8e f0  movl    %eax, -16(%rsi,%r9,4)
100000b11:      42 8b 44 8f f8  movl    -8(%rdi,%r9,4), %eax
100000b16:      83 c0 03        addl    $3, %eax
100000b19:      42 89 44 8e f4  movl    %eax, -12(%rsi,%r9,4)
100000b1e:      42 8b 44 8f fc  movl    -4(%rdi,%r9,4), %eax
100000b23:      83 c0 03        addl    $3, %eax
100000b26:      42 89 44 8e f8  movl    %eax, -8(%rsi,%r9,4)
100000b2b:      42 8b 04 8f     movl    (%rdi,%r9,4), %eax
100000b2f:      83 c0 03        addl    $3, %eax
100000b32:      42 89 44 8e fc  movl    %eax, -4(%rsi,%r9,4)
100000b37:      42 8b 44 8f 04  movl    4(%rdi,%r9,4), %eax
100000b3c:      83 c0 03        addl    $3, %eax
100000b3f:      42 89 04 8e     movl    %eax, (%rsi,%r9,4)
100000b43:      49 8d 41 09     leaq    9(%r9), %rax
100000b47:      49 ff c1        incq    %r9
100000b4a:      4d 39 c1        cmpq    %r8, %r9
100000b4d:      49 89 c1        movq    %rax, %r9
100000b50:      0f 8c 7a ff ff ff       jl      -134 <_do_loops+0x60>
100000b56:      83 f9 02        cmpl    $2, %ecx
100000b59:      7c 76           jl      118 <_do_loops+0x161>
100000b5b:      b8 08 00 00 00  movl    $8, %eax
```

D. Code

```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;
    }
}
```

(2) Loop Fusion

A. Performance observations

Argv / **Optimization**	**10000000**	**100000000**
-O2	12.596000ms	126.103000ms
-O3	12.182000ms	126.999000ms

B. Result analysis

It matches my expectation. Loop fusion does improve performance.

Reason: Apparently, it reduces one loop as well as overhead of loop management instructions. On the other hand, it also increases locality.

C. objdump snapshot

Instead of 3 loops, there are 2 loops in the assembly code below, showing the result of the loop fusion.

```
1000008f0:      8b 4c 87 04     movl    4(%rdi,%rax,4), %ecx
1000008f4:      83 c1 03        addl    $3, %ecx
1000008f7:      89 0c 86        movl    %ecx, (%rsi,%rax,4)
1000008fa:      8b 4c 86 fc     movl    -4(%rsi,%rax,4), %ecx
1000008fe:      83 c1 02        addl    $2, %ecx
100000901:      89 0c 82        movl    %ecx, (%rdx,%rax,4)
100000904:      8b 4c 87 08     movl    8(%rdi,%rax,4), %ecx
100000908:      83 c1 03        addl    $3, %ecx
10000090b:      89 4c 86 04     movl    %ecx, 4(%rsi,%rax,4)
10000090f:      8b 0c 86        movl    (%rsi,%rax,4), %ecx
100000912:      83 c1 02        addl    $2, %ecx
100000915:      89 4c 82 04     movl    %ecx, 4(%rdx,%rax,4)
100000919:      48 8d 40 02     leaq    2(%rax), %rax
10000091d:      49 39 c0        cmpq    %rax, %r8
100000920:      75 ce   jne     -50 <_do_loops+0x140>
100000922:      45 85 c9        testl   %r9d, %r9d
100000925:      0f 84 8e 00 00 00       je      142 <_do_loops+0x209>
10000092b:      8b 4c 87 04     movl    4(%rdi,%rax,4), %ecx
10000092f:      83 c1 03        addl    $3, %ecx
100000932:      89 0c 86        movl    %ecx, (%rsi,%rax,4)
100000935:      8b 4c 86 fc     movl    -4(%rsi,%rax,4), %ecx
100000939:      83 c1 02        addl    $2, %ecx
10000093c:      89 0c 82        movl    %ecx, (%rdx,%rax,4)
10000093f:      eb 78   jmp     120 <_do_loops+0x209>
100000941:      8b 06   movl    (%rsi), %eax
100000943:      41 89 ca        movl    %ecx, %r10d
100000946:      41 f7 d2        notl    %r10d
100000949:      41 83 e2 01     andl    $1, %r10d
10000094d:      41 b9 01 00 00 00       movl    $1, %r9d
100000953:      83 f9 02        cmpl    $2, %ecx
100000956:      74 49   je      73 <_do_loops+0x1f1>
100000958:      44 89 d1        movl    %r10d, %ecx
10000095b:      49 29 c8        subq    %rcx, %r8
10000095e:      41 b9 01 00 00 00       movl    $1, %r9d
100000964:      66 2e 0f 1f 84 00 00 00 00 00   nopw    %cs:(%rax,%rax)
10000096e:      66 90   nop
100000970:      42 8b 4c 8f 04  movl    4(%rdi,%r9,4), %ecx
100000975:      8d 59 03        leal    3(%rcx), %ebx
100000978:      42 89 1c 8e     movl    %ebx, (%rsi,%r9,4)
10000097c:      83 c0 02        addl    $2, %eax
10000097f:      42 89 04 8a     movl    %eax, (%rdx,%r9,4)
100000983:      42 8b 44 8f 08  movl    8(%rdi,%r9,4), %eax
100000988:      83 c0 03        addl    $3, %eax
10000098b:      42 89 44 8e 04  movl    %eax, 4(%rsi,%r9,4)
100000990:      83 c1 05        addl    $5, %ecx
100000993:      42 89 4c 8a 04  movl    %ecx, 4(%rdx,%r9,4)
100000998:      4d 8d 49 02     leaq    2(%r9), %r9
10000099c:      4d 39 c8        cmpq    %r9, %r8
10000099f:      75 cf   jne     -49 <_do_loops+0x1c0>
1000009a1:      45 85 d2        testl   %r10d, %r10d
1000009a4:      74 13   je      19 <_do_loops+0x209>
```

D. Code

```
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;
    }
}
```

(3) Loop Reversal

A. Performance observations

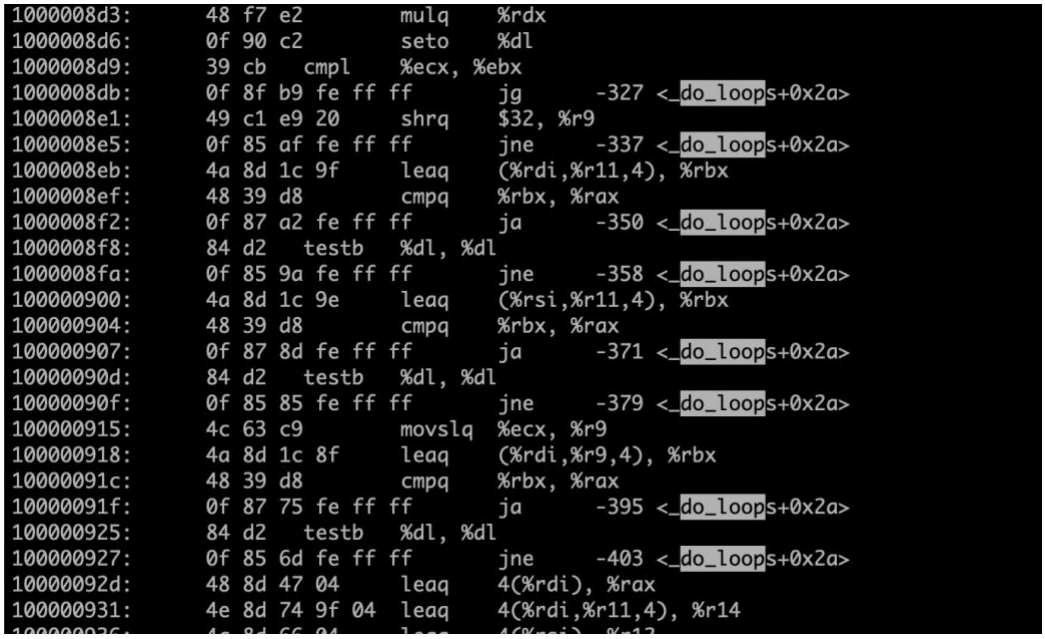
Argv / Optimization	10000000	100000000
-O2	16.413000ms	161.148000ms
-O3	15.686000ms	159.115000ms

B. Result analysis

It doesn't match my expectation. Loop reversal does not improve performance for O2, which is not what I expect.

Reason: For O2, the compiler may be thinking it's doing something smart that's not quite working out well for this particular code. See next part for more details.

C. objdump snapshot



```
1000008d3:      48 f7 e2        mulq    %rdx
1000008d6:      0f 90 c2        seto    %dl
1000008d9:      39 cb   cmpl    %ecx, %ebx
1000008db:      0f 8f b9 fe ff ff       jg      -327 <_do_loops+0x2a>
1000008e1:      49 c1 e9 20     shrq    $32, %r9
1000008e5:      0f 85 af fe ff ff       jne     -337 <_do_loops+0x2a>
1000008eb:      4a 8d 1c 9f     leaq    (%rdi,%r11,4), %rbx
1000008ef:      48 39 d8        cmpq    %rbx, %rax
1000008f2:      0f 87 a2 fe ff ff       ja      -350 <_do_loops+0x2a>
1000008f8:      84 d2   testb   %dl, %dl
1000008fa:      0f 85 9a fe ff ff       jne     -358 <_do_loops+0x2a>
100000900:      4a 8d 1c 9e     leaq    (%rsi,%r11,4), %rbx
100000904:      48 39 d8        cmpq    %rbx, %rax
100000907:      0f 87 8d fe ff ff       ja      -371 <_do_loops+0x2a>
10000090d:      84 d2   testb   %dl, %dl
10000090f:      0f 85 85 fe ff ff       jne     -379 <_do_loops+0x2a>
100000915:      4c 63 c9        movslq  %ecx, %r9
100000918:      4a 8d 1c 8f     leaq    (%rdi,%r9,4), %rbx
10000091c:      48 39 d8        cmpq    %rbx, %rax
10000091f:      0f 87 75 fe ff ff       ja      -395 <_do_loops+0x2a>
100000925:      84 d2   testb   %dl, %dl
100000927:      0f 85 6d fe ff ff       jne     -403 <_do_loops+0x2a>
10000092d:      48 8d 47 04     leaq    4(%rdi), %rax
100000931:      4e 8d 74 9f 04  leaq    4(%rdi,%r11,4), %r14
```

The disassembly code is more complex than a simple loop fusion, which includes a lot of "jump" instructions, branching to the beginning of *do_loops* function stack. Some examples are shown above. Probably, this can be a cause to slow down the performance by O2.

D. Code

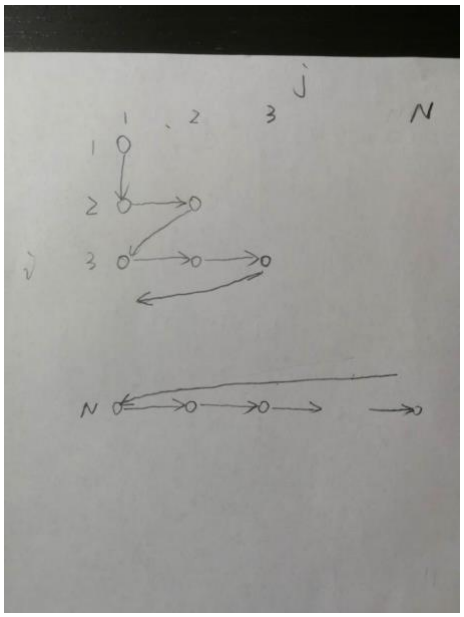
```
void do_loops(int *a, int *b, int *c, int N) {
   int i;
   for (i = N - 1; i >= 1; i--) {
      a[i] = a[i] + 1;
      b[i] = a[i + 1] + 3;
   }
   for (i = 1; i < N; i++) {
      c[i] = b[i - 1] + 2;
   }
}
```

(d)

I can beat the compiler. Based on (c), I can even beat it by implementing a single loop transformation: loop fusion. I can even beat it more by combining several loop transformations. It is because, on the one hand, there are cases where a compiler may not be able to apply an optimization; On the other hand, some of these transformations, like loop peeling and loop strip mining, are not as easy (or even impossible) for a compiler to implement. Therefore, a hand-tuned code can beat a compiler.

2 Dependence Analysis

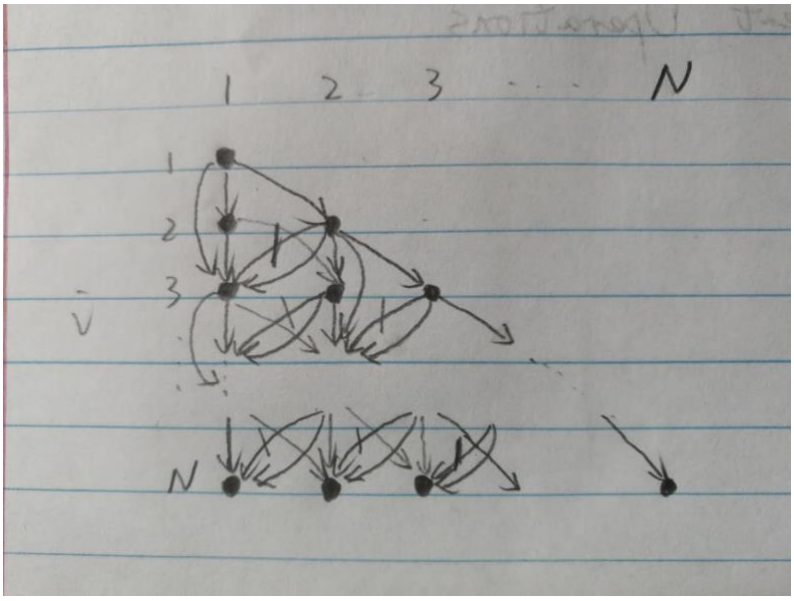
(a)



(b)

(1) S1[i, j] ➔ A S1[i + 1, j - 1] (loop-carried)
(2) S1[i, j] ➔ A S2[i, j] (loop-independent)
(3) S1[i, j] ➔ T S2[i + 1, j + 1] (loop-carried)
(4) S3[i, j] ➔ T S1[i + 1, j] (loop-carried)
(5) S1[i, j] ➔ T S3[i, j] (loop-independent)
(6) S4[i, j] ➔ T S4[i + 1, j - 1] (loop-carried)
(7) S3[i, j] ➔ T S2[i + 2, j] (loop-carried)

(c)



3 Function In-lining and Performance

(a)

Argv / Inline or not	10000000	100000000
noinline	11.816ms	116.291ms
-always_inline	6.141ms	63.946ms

*The recorded time is the shortest time seen in a test of 8 each (same in the following tables).

(b)

Always inline:

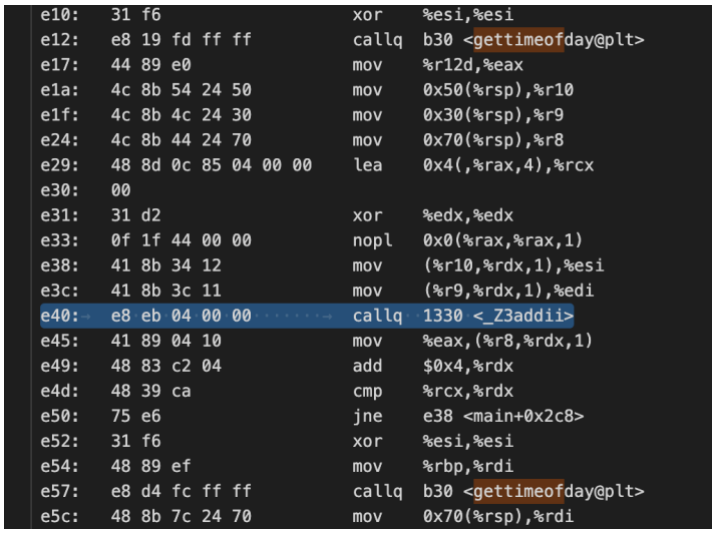
```
e10:        31 f6                   xor    %esi,%esi
e12:        e8 19 fd ff ff          callq  b30 <gettimeofday@plt>
e17:        48 8b 7c 24 50          mov    0x50(%rsp),%rdi
e1c:        48 8b 74 24 70          mov    0x70(%rsp),%rsi
e21:        4c 8b 44 24 30          mov    0x30(%rsp),%r8
e26:        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>
f76: 4c 8b 4c 24 70      mov    0x70(%rsp),%r9
```

From the assembly code snippet of the main function, I highlight two lines, indicating the function call to *gettimeofday*. Between these two lines, we find that there is no other *callq* instructions. Therefore, function in-lining doesn't make a call stack, instead, it runs the array addition loop inside of the main function.

No-line:



Obviously from the screenshot above, explicitly asking the compiler not to inline the *add* function creates a function call to addition. (The screenshot is a snippet of main function).

(c)

The results match my expectations. Because function in-lining reduces overhead instructions such as saving and restoring the call stack, thus, it reduces the overall number of instructions. Also, it gets rid of function call and return instructions. Lastly, it may allow the compiler to better optimize the code. As a consequence, the performance is improved by function in-lining.

(d)

Argv / Inline or not	10000000	100000000
No attribute specified	6.286ms	63.801ms

The performance is the same as in-lining results. And I also inspect the assembly code: it doesn't call *add* function in *main*, either. Therefore, I think the compiler O3 is in-lining the *add()* function by default.

4 Loop transformations

(1) Loop Invariant Hoisting

```
/************************************
 *      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;
        }
    }
}
...
```

(2) Loop Unswitching

```
/*************************************
 *         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;
        }
    }
}
...
```

(3) Loop Interchange

```
/*************************************
 *         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];
        }
    }
}
...
```

(4) Loop Unrolling

```
/************************************
 *         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] + 1;
            sum = sum + a[j][1] + 1;
            sum = sum + a[j][2] + 1;
            sum = sum + a[j][3] + 1;
    }
}
...
```

(5) Other (Loop Peeling and Loop Strip Mining)
If we are provided more details, like cache size and memory alignment, we are also able to apply loop peeling as well as loop strip mining.

(6) Final code

```
...
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] + 1;
            sum = sum + a[j][1] + 1;
            sum = sum + a[j][2] + 1;
            sum = sum + a[j][3] + 1;
    }
}
...
```

5 Loop transformations

(a) Loop fusion

It is unsafe.

Code status / Dependencies	Original	After transformation
S1 and S3	S1➔O S3 (loop-carried)	S3 ➔ O S1(loop-carried)
S2 and S3	S2 ➔ A S3(loop-carried)	S3 ➔ T S2(loop-carried)

Loop Fusion is safe iff no data dependence between the nests becomes loop-carried data dependence of a different type. Since this condition is violated by the dependence between S2 and S3, the loop transformation is not safe.

(b) Loop interchange

It is unsafe.

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 $j < j'$). However, the given example carries such data dependence where $i < i'$ and $j > j'$. Therefore, the loop transformation is not safe.

(c) Loop fission

It is safe.