ECE 565 - HW1

Yi Mi

ym154@duke.edu

September 2, 2020

1. Code Optimization

(a)

- Compile and run the unmodified base code, and record performance. I used N = 100000000 and N = 10000000, and ran 5 times for -O2 and -O3.

-O2

```
gcc –O2 –o loop_performance loop_performance.c
./loop_performance 100000000 / 10000000
```

-O3

```
gcc –O3 –o loop_performance loop_performance.c
./loop_performance 100000000 / 10000000
```

- Performance

N = 100000000

optimization level	1	2	3	4	5	average	shortest
-O2	224.213	266.429	225.184	223.547	253.688	238.612	223.547
-O3	187.985	186.976	190.266	188.705	197.857	190.358	186.976

sum = 699999988

N = 10000000

optimization level	1	2	3	4	5	average	shortest
-O2	22.149	23.277	22.579	22.385	21.966	22.4712	21.966
-O3	19.095	18.585	18.518	18.544	18.847	18.7178	18.518

sum = 69999988

(b)

- Processor architecture: x86_64

- CPU frequency: 2.30GHz
- OS type: Ubuntu18 Server
- VM or standalone system: VM

(c)

- I use the command line below to execute. All the assembly code is based on -O2 optimization level.

```
gcc -O2 -o loop_performance loop_performance.c
./loop_performance 100000000
objdump -d loop_performance
```

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

- Original assembly code (complete)

```
0000000000000a70 <do_loops>:
 a70:   8d 41 ff                lea    -0x1(%rcx),%eax
 a73:   85 c0                   test   %eax,%eax
 a75:   7e 2d                   jle    aa4 <do_loops+0x34>
 a77:   48 98                   cltq
 a79:   44 8d 49 fe             lea    -0x2(%rcx),%r9d
 a7d:   4c 8d 04 85 00 00 00    lea    0x0(,%rax,4),%r8
 a84:   00
 a85:   49 c1 e1 02             shl    $0x2,%r9
 a89:   4a 8d 04 07             lea    (%rdi,%r8,1),%rax
 a8d:   4e 8d 44 07 fc          lea    -0x4(%rdi,%r8,1),%r8
 a92:   4d 29 c8                sub    %r9,%r8
 a95:   0f 1f 00                nopl   (%rax)
 a98:   83 00 01                addl   $0x1,(%rax)
 a9b:   48 83 e8 04             sub    $0x4,%rax
 a9f:   4c 39 c0                cmp    %r8,%rax
 aa2:   75 f4                   jne    a98 <do_loops+0x28>
 aa4:   83 f9 01                cmp    $0x1,%ecx
 aa7:   7e 4a                   jle    af3 <do_loops+0x83>
 aa9:   8d 41 fe                lea    -0x2(%rcx),%eax
 aac:   4c 8d 04 85 04 00 00    lea    0x4(,%rax,4),%r8
 ab3:   00
 ab4:   31 c0                   xor    %eax,%eax
 ab6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 abd:   00 00 00
 ac0:   8b 4c 07 08             mov    0x8(%rdi,%rax,1),%ecx
 ac4:   83 c1 03                add    $0x3,%ecx
 ac7:   89 4c 06 04             mov    %ecx,0x4(%rsi,%rax,1)
 acb:   48 83 c0 04             add    $0x4,%rax
 acf:   49 39 c0                cmp    %rax,%r8
 ad2:   75 ec                   jne    ac0 <do_loops+0x50>
 ad4:   31 c9                   xor    %ecx,%ecx
 ad6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 add:   00 00 00
 ae0:   8b 3c 0e                mov    (%rsi,%rcx,1),%edi
 ae3:   83 c7 02                add    $0x2,%edi
 ae6:   89 7c 0a 04             mov    %edi,0x4(%rdx,%rcx,1)
 aea:   48 83 c1 04             add    $0x4,%rcx
 aee:   48 39 c1                cmp    %rax,%rcx
 af1:   75 ed                   jne    ae0 <do_loops+0x70>
 af3:   f3 c3                   repz retq
 af5:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 afc:   00 00 00
 aff:   90                      nop
```

i. Loop Invariant Hoisting

- Not applicable for this question.

ii. Loop Unrolling

- 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+=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++) {
      c[i] = b[i-1] + 2;
  }
}
```

- Performance

N = 100000000

optimization level	1	2	3	4	5	average	shortest
-O2	223.647	217.736	218.346	219.578	219.313	219.724	217.736
-O3	189.113	7+I13:N14	186.615	188.152	199.478	190.84	186.615

sum = 699999988

N = 10000000

optimization level	1	2	3	4	5	average	shortest
-O2	21.752	28.506	22.156	22.101	26.638	24.2306	21.752
-O3	19.935	18.426	18.494	18.621	18.515	18.7982	18.426

sum = 69999988

With N equals to 100,000,000, for -O2 optimization, my code improve a little of the performance, but it is almost the same for -O3 optimization. It match my expectation. With N equals to 10,000,000, they are similar.

- Assembly code (Part of)

```
0000000000000a70 <do_loops>:
 a70:   8d 41 ff                lea    -0x1(%rcx),%eax
 a73:   85 c0                   test   %eax,%eax
 a75:   7e 2d                   jle    aa4 <do_loops+0x34>
 a77:   48 98                   cltq
 a79:   44 8d 49 fe             lea    -0x2(%rcx),%r9d
 a7d:   4c 8d 04 85 00 00 00    lea    0x0(,%rax,4),%r8
 a84:   00
 a85:   49 c1 e1 02             shl    $0x2,%r9
 a89:   4a 8d 04 07             lea    (%rdi,%r8,1),%rax
 a8d:   4e 8d 44 07 fc          lea    -0x4(%rdi,%r8,1),%r8
 a92:   4d 29 c8                sub    %r9,%r8
 a95:   0f 1f 00                nopl   (%rax)
 a98:   83 00 01                addl   $0x1,(%rax)
 a9b:   48 83 e8 04             sub    $0x4,%rax
 a9f:   4c 39 c0                cmp    %r8,%rax
 aa2:   75 f4                   jne    a98 <do_loops+0x28>
 aa4:   83 f9 01                cmp    $0x1,%ecx
 aa7:   7e 7a                   jle    b23 <do_loops+0xb3>
 aa9:   83 e9 02                sub    $0x2,%ecx
 aac:   48 8d 47 08             lea    0x8(%rdi),%rax
 ab0:   4c 8d 46 04             lea    0x4(%rsi),%r8
 ab4:   41 89 c9                mov    %ecx,%r9d
 ab7:   41 c1 e9 02             shr    $0x2,%r9d
 abb:   49 c1 e1 04             shl    $0x4,%r9
 abf:   4e 8d 4c 0f 18          lea    0x18(%rdi,%r9,1),%r9
 ac4:   0f 1f 40 00             nopl   0x0(%rax)
 ac8:   8b 38                   mov    (%rax),%edi
 aca:   48 83 c0 10             add    $0x10,%rax
 ace:   49 83 c0 10             add    $0x10,%r8
 ad2:   83 c7 03                add    $0x3,%edi
 ad5:   41 89 78 f0             mov    %edi,-0x10(%r8)
 ad9:   8b 78 f4                mov    -0xc(%rax),%edi
 adc:   83 c7 03                add    $0x3,%edi
 adf:   41 89 78 f4             mov    %edi,-0xc(%r8)
 ae3:   8b 78 f8                mov    -0x8(%rax),%edi
 ae6:   83 c7 03                add    $0x3,%edi
 ae9:   41 89 78 f8             mov    %edi,-0x8(%r8)
 aed:   8b 78 fc                mov    -0x4(%rax),%edi
 af0:   83 c7 03                add    $0x3,%edi
 af3:   41 89 78 fc             mov    %edi,-0x4(%r8)
 af7:   4c 39 c8                cmp    %r9,%rax
 afa:   75 cc                   jne    ac8 <do_loops+0x58>
```

Based on the assembly code, we can see that in the second for-loop there are more instructions inside, since we unroll the code. The reason of performance improvement is possibly additional ILP and instruction count reduction. And Loop unrolling might be more suitable for bigger N.

iii. Loop Fusion

- 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;
  }
}
```

- Performance

N = 100000000

optimization level	1	2	3	4	5	average	shortest
-O2	210.574	207.764	213.344	216.591	214.357	212.526	207.764
-O3	224.76	224.363	225.331	225.653	241.842	228.39	224.363

sum = 699999988

N = 10000000

optimization level	1	2	3	4	5	average	shortest
-O2	22.161	21.794	21.223	21.399	21.642	21.6438	21.223
-O3	22.675	22.653	22.861	26.346	22.524	23.4118	22.524

sum = 69999988

With N equals to 100,000,000, for -O2 optimization, my code improves about 25 milliseconds of the performance, but it is even worse for -O3 optimization. With N equals to 10,000,000, the results of -O2 are similar, but it is also worse for -O3 optimization. It did not match my expectation.

- Assembly code (complete)

```
0000000000000a70 <do_loops>:
 a70:   8d 41 ff                lea    -0x1(%rcx),%eax
 a73:   85 c0                   test   %eax,%eax
 a75:   7e 2d                   jle    aa4 <do_loops+0x34>
 a77:   48 98                   cltq
 a79:   44 8d 49 fe             lea    -0x2(%rcx),%r9d
 a7d:   4c 8d 04 85 00 00 00    lea    0x0(,%rax,4),%r8
 a84:   00
 a85:   49 c1 e1 02             shl    $0x2,%r9
 a89:   4a 8d 04 07             lea    (%rdi,%r8,1),%rax
 a8d:   4e 8d 44 07 fc          lea    -0x4(%rdi,%r8,1),%r8
 a92:   4d 29 c8                sub    %r9,%r8
 a95:   0f 1f 00                nopl   (%rax)
 a98:   83 00 01                addl   $0x1,(%rax)
 a9b:   48 83 e8 04             sub    $0x4,%rax
 a9f:   4c 39 c0                cmp    %r8,%rax
 aa2:   75 f4                   jne    a98 <do_loops+0x28>
 aa4:   83 f9 01                cmp    $0x1,%ecx
 aa7:   7e 34                   jle    add <do_loops+0x6d>
 aa9:   8d 41 fe                lea    -0x2(%rcx),%eax
 aac:   4c 8d 04 85 08 00 00    lea    0x8(,%rax,4),%r8
 ab3:   00
 ab4:   b8 04 00 00 00          mov    $0x4,%eax
 ab9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
 ac0:   8b 4c 07 04             mov    0x4(%rdi,%rax,1),%ecx
 ac4:   83 c1 03                add    $0x3,%ecx
 ac7:   89 0c 06                mov    %ecx,(%rsi,%rax,1)
 aca:   8b 4c 06 fc             mov    -0x4(%rsi,%rax,1),%ecx
 ace:   83 c1 02                add    $0x2,%ecx
 ad1:   89 0c 02                mov    %ecx,(%rdx,%rax,1)
 ad4:   48 83 c0 04             add    $0x4,%rax
 ad8:   4c 39 c0                cmp    %r8,%rax
 adb:   75 e3                   jne    ac0 <do_loops+0x50>
 add:   f3 c3                   repz retq
 adf:   90                      nop
```

The assembly code is shorter than the original code. It executes two lines of code inside a loop. The reason that the performance of -O2 optimization level is better might be that loop fusion reduces overhead of loop management instructions, increases granularity of work done in a loop, or improves data locality. The reason that the performance of -O3 optimization level is worse is it's possible that a code change causes the compiler for some reason not to generate code that performs as well as the original code.

iv. Loop Peeling

- Code

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

```
    if (N-1>=1){
        a[N-1] = a[N-1] + 1;
    }
    for (i=N-2; 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;
    }
}
```

- Performance

  N = 100000000

optimization level	1	2	3	4	5	average	shortest
-O2	228.199	226.783	227.192	225.961	225.566	226.74	225.566
-O3	188.067	189.839	190.505	191.661	188.927	189.8	188.067

  sum = 699999988

  N = 10000000

optimization level	1	2	3	4	5	average	shortest
-O2	22.346	22.271	22.483	22.39	24.896	22.8772	22.271
-O3	20.887	18.689	18.696	18.612	19.085	19.1938	18.612

  sum = 69999988

  With N equals to 100,000,000, for -O2 optimization, my code improve about 10 milliseconds of the performance, and it is roughly the same for -O3 optimization. It match my expectation. With N equals to 10,000,000, they are similar.

- Assembly code (part of)

```
0000000000000a70 <do_loops>:
 a70:   83 f9 01                cmp    $0x1,%ecx
 a73:   7e 08                   jle    a7d <do_loops+0xd>
 a75:   48 63 c1                movslq %ecx,%rax
 a78:   83 44 87 fc 01          addl   $0x1,-0x4(%rdi,%rax,4)
 a7d:   44 8d 49 fe             lea    -0x2(%rcx),%r9d
 a81:   45 85 c9                test   %r9d,%r9d
 a84:   7e 2e                   jle    ab4 <do_loops+0x44>
 a86:   4d 63 c1                movslq %r9d,%r8
 a89:   44 8d 51 fd             lea    -0x3(%rcx),%r10d
 a8d:   49 c1 e0 02             shl    $0x2,%r8
 a91:   4a 8d 04 07             lea    (%rdi,%r8,1),%rax
 a95:   4e 8d 44 07 fc          lea    -0x4(%rdi,%r8,1),%r8
 a9a:   49 c1 e2 02             shl    $0x2,%r10
 a9e:   4d 29 d0                sub    %r10,%r8
 aa1:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
 aa8:   83 00 01                addl   $0x1,(%rax)
 aab:   48 83 e8 04             sub    $0x4,%rax
 aaf:   4c 39 c0                cmp    %r8,%rax
 ab2:   75 f4                   jne    aa8 <do_loops+0x38>
 ab4:   83 f9 01                cmp    $0x1,%ecx
 ab7:   7e 4a                   jle    b03 <do_loops+0x93>
 ab9:   45 89 c9                mov    %r9d,%r9d
 abc:   31 c0                   xor    %eax,%eax
 abe:   4e 8d 04 8d 04 00 00    lea    0x4(,%r9,4),%r8
 ac5:   00
 ac6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 acd:   00 00 00
 ad0:   8b 4c 07 08             mov    0x8(%rdi,%rax,1),%ecx
 ad4:   83 c1 03                add    $0x3,%ecx
 ad7:   89 4c 06 04             mov    %ecx,0x4(%rsi,%rax,1)
 adb:   48 83 c0 04             add    $0x4,%rax
 adf:   49 39 c0                cmp    %rax,%r8
 ae2:   75 ec                   jne    ad0 <do_loops+0x60>
 ae4:   31 c9                   xor    %ecx,%ecx
 ae6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 aed:   00 00 00
 af0:   8b 3c 0e                mov    (%rsi,%rcx,1),%edi
 af3:   83 c7 02                add    $0x2,%edi
 af6:   89 7c 0a 04             mov    %edi,0x4(%rdx,%rcx,1)
 afa:   48 83 c1 04             add    $0x4,%rcx
 afe:   48 39 c1                cmp    %rax,%rcx
 b01:   75 ed                   jne    af0 <do_loops+0x80>
 b03:   f3 c3                   repz retq
```

We can see that there are comparison and addition instructions before the for-loop. The performance increases might because that it enforces a memory alignment on array references. And Loop unrolling might be more suitable for bigger N.

v. Loop Unswitching:

- Not applicable.

vi. Loop interchange:

- Not applicable.

vii. Loop Reversal

- Code

```
void do_loops(int *a, int *b, int *c, int  N){
  int i;
  for (i=1; i<N; 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;
  }
}
```

- Performance

N = 100000000

optimization level	1	2	3	4	5	average	shortest
-O2	211.864	211.402	213.391	229.445	214.55	216.13	211.402
-O3	176.262	183.373	196.692	179.525	176.065	182.383	176.065

sum = 699999988

N = 10000000

optimization level	1	2	3	4	5	average	shortest
-O2	27.355	21.539	21.784	21.573	21.816	22.8134	21.539
-O3	17.206	17.254	17.509	17.441	17.695	17.421	17.206

sum = 69999988

With N equals to 100,000,000, for -O2 optimization, my code improve about 20 milliseconds of the performance, and it is improve about 10 milliseconds of the performance for -O3 optimization. It match my expectation. With N equals to 10,000,000,

they are similar.

- Assembly code (complete)

```
0000000000000a70 <do_loops>:
 a70:   83 f9 01                cmp    $0x1,%ecx
 a73:   7e 5e                   jle    ad3 <do_loops+0x63>
 a75:   44 8d 41 fe             lea    -0x2(%rcx),%r8d
 a79:   48 8d 47 04             lea    0x4(%rdi),%rax
 a7d:   4a 8d 4c 87 08          lea    0x8(%rdi,%r8,4),%rcx
 a82:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
 a88:   83 00 01                addl   $0x1,(%rax)
 a8b:   48 83 c0 04             add    $0x4,%rax
 a8f:   48 39 c8                cmp    %rcx,%rax
 a92:   75 f4                   jne    a88 <do_loops+0x18>
 a94:   4e 8d 04 85 04 00 00    lea    0x4(,%r8,4),%r8
 a9b:   00
 a9c:   31 c0                   xor    %eax,%eax
 a9e:   66 90                   xchg   %ax,%ax
 aa0:   8b 4c 07 08             mov    0x8(%rdi,%rax,1),%ecx
 aa4:   83 c1 03                add    $0x3,%ecx
 aa7:   89 4c 06 04             mov    %ecx,0x4(%rsi,%rax,1)
 aab:   48 83 c0 04             add    $0x4,%rax
 aaf:   49 39 c0                cmp    %rax,%r8
 ab2:   75 ec                   jne    aa0 <do_loops+0x30>
 ab4:   31 c9                   xor    %ecx,%ecx
 ab6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 abd:   00 00 00
 ac0:   8b 3c 0e                mov    (%rsi,%rcx,1),%edi
 ac3:   83 c7 02                add    $0x2,%edi
 ac6:   89 7c 0a 04             mov    %edi,0x4(%rdx,%rcx,1)
 aca:   48 83 c1 04             add    $0x4,%rcx
 ace:   48 39 c1                cmp    %rax,%rcx
 ad1:   75 ed                   jne    ac0 <do_loops+0x50>
 ad3:   f3 c3                   repz retq
 ad5:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 adc:   00 00 00
 adf:   90                      nop
```

The assembly code is shorter than the original one. And Loop reversal might be more suitable for bigger N.

viii. Loop Unroll and Jam

- Not applicable.

ix. Loop Strip Mining

- 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 (j=1; j<N; j+=1000) {
      for (i=j; i<(j+1000);i++){
      b[i] = a[i+1] + 3;
}


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

- Performance

  N = 100000000

optimization level	1	2	3	4	5	average	shortest
-O2	222.813	230.297	221.424	224.332	221.518	224.077	221.424
-O3	187.415	185.982	186.319	186.944	186.051	186.542	185.982

  sum = 699999988

  N = 10000000

optimization level	1	2	3	4	5	average	shortest
-O2	23.031	21.929	21.915	22.235	22.029	22.2278	21.915
-O3	18.505	18.525	18.341	21.079	18.343	18.9586	18.341

  sum = 699999988

  With N equals to 100,000,000, for -O2 optimization, my code improve about 14 milliseconds of the performance, and it is improve about 4 milliseconds of the performance for -O3 optimization. It match my expectation. With N equals to 10,000,000, they are similar.

- Assembly code (part of)

```
0000000000000a70 <do_loops>:
 a70:   8d 41 ff                lea    -0x1(%rcx),%eax
 a73:   85 c0                   test   %eax,%eax
 a75:   7e 2d                   jle    aa4 <do_loops+0x34>
 a77:   48 98                   cltq
 a79:   44 8d 49 fe             lea    -0x2(%rcx),%r9d
 a7d:   4c 8d 04 85 00 00 00    lea    0x0(,%rax,4),%r8
 a84:   00
 a85:   49 c1 e1 02             shl    $0x2,%r9
 a89:   4a 8d 04 07             lea    (%rdi,%r8,1),%rax
 a8d:   4e 8d 44 07 fc          lea    -0x4(%rdi,%r8,1),%r8
 a92:   4d 29 c8                sub    %r9,%r8
 a95:   0f 1f 00                nopl   (%rax)
 a98:   83 00 01                addl   $0x1,(%rax)
 a9b:   48 83 e8 04             sub    $0x4,%rax
 a9f:   4c 39 c0                cmp    %r8,%rax
 aa2:   75 f4                   jne    a98 <do_loops+0x28>
 aa4:   83 f9 01                cmp    $0x1,%ecx
 aa7:   7e 7a                   jle    b23 <do_loops+0xb3>
 aa9:   4c 8d 47 04             lea    0x4(%rdi),%r8
 aad:   4c 8d 4e 04             lea    0x4(%rsi),%r9
 ab1:   41 ba 01 00 00 00       mov    $0x1,%r10d
 ab7:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
 abe:   00 00
 ac0:   31 c0                   xor    %eax,%eax
 ac2:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
 ac8:   41 8b 7c 00 04          mov    0x4(%r8,%rax,1),%edi
 acd:   83 c7 03                add    $0x3,%edi
 ad0:   41 89 3c 01             mov    %edi,(%r9,%rax,1)
 ad4:   48 83 c0 04             add    $0x4,%rax
 ad8:   48 3d a0 0f 00 00       cmp    $0xfa0,%rax
 ade:   75 e8                   jne    ac8 <do_loops+0x58>
 ae0:   41 81 c2 e8 03 00 00    add    $0x3e8,%r10d
 ae7:   49 81 c0 a0 0f 00 00    add    $0xfa0,%r8
 aee:   49 81 c1 a0 0f 00 00    add    $0xfa0,%r9
 af5:   44 39 d1                cmp    %r10d,%ecx
 af8:   7f c6                   jg     ac0 <do_loops+0x50>
 afa:   8d 41 fe                lea    -0x2(%rcx),%eax
 afd:   48 8d 3c 85 04 00 00    lea    0x4(,%rax,4),%rdi
 b04:   00
 b05:   31 c0                   xor    %eax,%eax
 b07:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
 b0e:   00 00
```

The assembly is longer than the original one since I created a nested for-loop. The reason that it increases the performance might be making vectorization easier. And Loop stripe mining might be more suitable for bigger N.

x. Loop Tiling

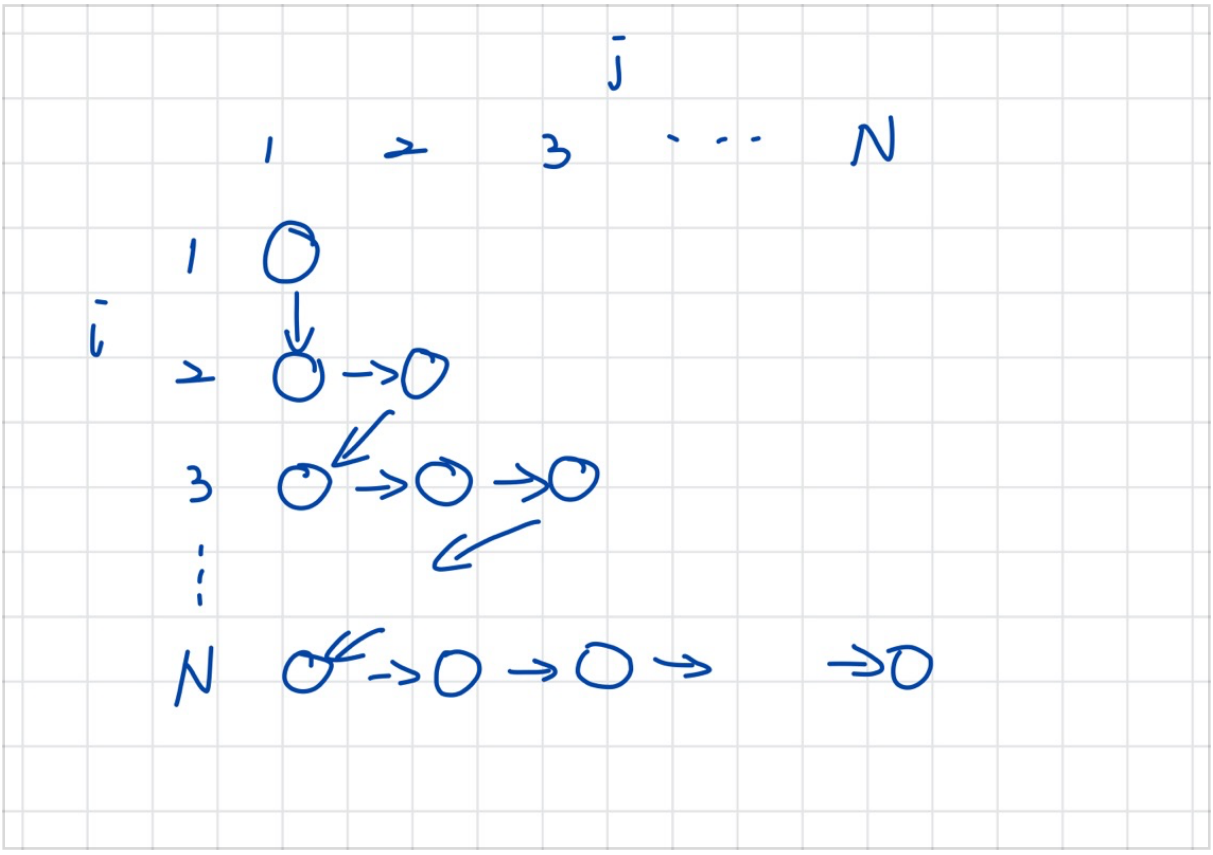
- Not applicable.

(d)

Yes. I think I beat the complier in some cases where I get better performance with my hand-tuned code, though with N equals to 10,000,000, the performance did not improve much compared to the situation N equals to 100,000,000.

2. Dependence Analysis

(a) ITG



(b) Dependency

S1: a[i][j] = b[i][j] + c[i][j] * a[i+1][j-1];

S2: b[i][j] = a[i-1][j-1] * c[i-1][j];

S3: c[i+1][j] = a[i][j];

S4: d[i][j] = d[i-1][j+1];

Number examples:

i = 1, j = 1

a[1][1] = b[1][1] + c[1][1] * a[2][0];

b[1][1] = a[0][0] * c[0][1];

c[2][1] = a[1][1];

d[1][1] = d[0][2];

i = 2, j = 1

a[2][1] = b[2][1] + c[2][1] * a[3][0];

b[2][1] = a[1][0] * c[1][1];

c[3][1] = a[2][1];

d[2][1] = d[1][2];

i = 2, j = 2

a[2][2] = b[2][2] + c[2][2] * a[3][1];

b[2][2] = a[1][1] * c[1][2];

c[3][2] = a[2][2];

d[2][2] = d[1][3];

i = 3, j = 1

a[3][1] = b[3][1] + c[3][1] * a[4][0];

b[3][1] = a[2][0] * c[2][1];

c[4][1] = a[3][1];

d[3][1] = d[2][2];

… …

Dependence:

① S1[i, j] ➜ A S1[i+1, j-1] (loop-carried)

② S1[i, j] ➜ T S2[i+1, j+1] (loop-carried)

③ S1[i, j] ➜ A S2[i, j] (loop-independent)

④ S1[i, j] ➜ T S3[i, j] (loop-independent)

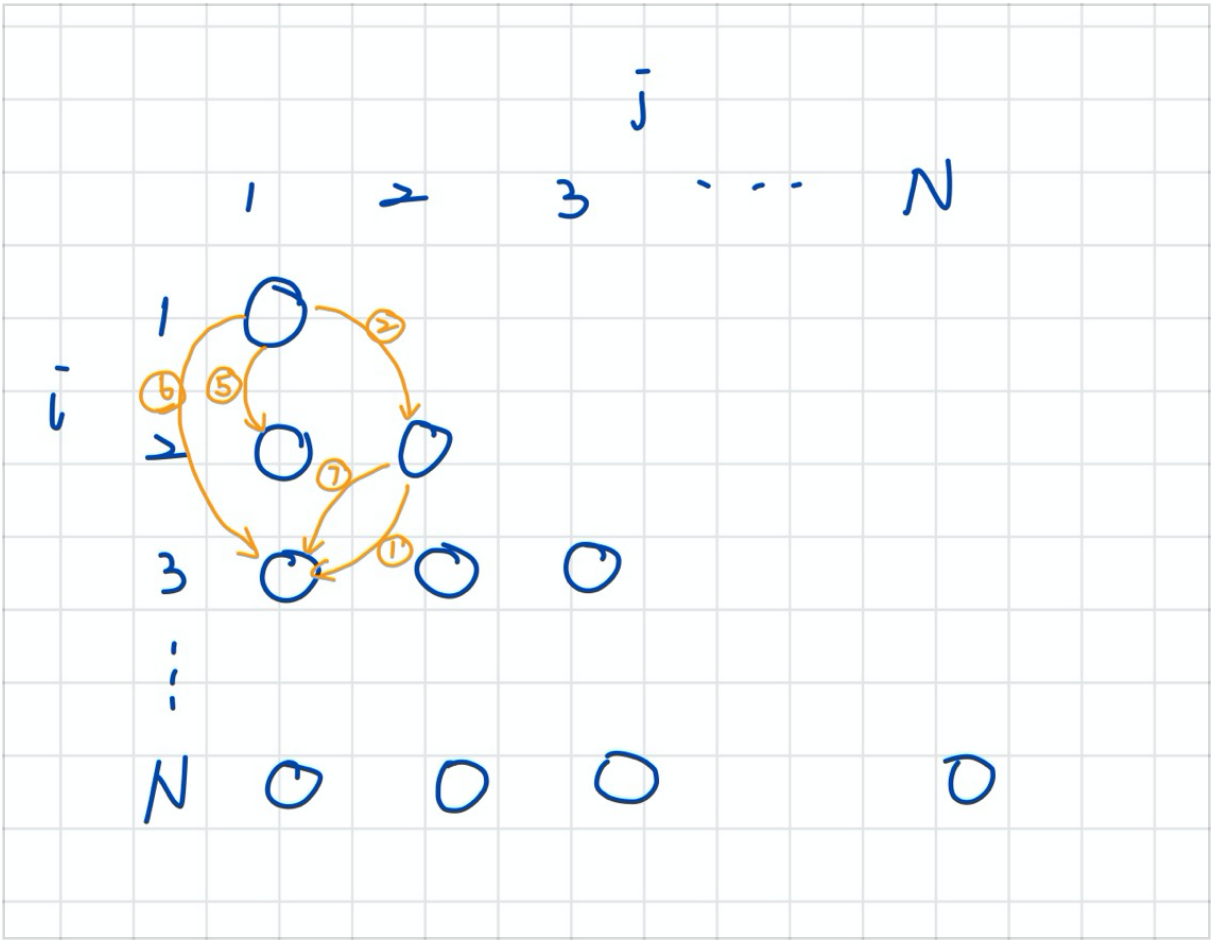
⑤ S3[i, j] ➜ T S1[i+1, j] (loop-carried)

⑥ S3[i, j] ➜ T S2[i+2, j] (loop-carried)

⑦ S4[i, j] ➜ T S4[i+1, j-1] (loop-carried)

(c) LDG

It would be so messy if I draw every edge on the LDG that I just draw one example edge of each dependence, marked with its serial number.



3. Function In-lining and Performance

(a)

- I used the command line below to execute.

```
g++ -O3 -o func_inlining func_inlining.cpp
./func_inlining 100000000
objdump -d func_inlining
```

inline attribute

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

not inline attribute

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

- Performance

	1	2	3	4	5	average
inline	96.376	97.325	101.906	98.325	100.958	98.978
not inline	188.414	189.119	187.68	188.495	186.244	187.99

checksum=-2120047872

(b)

- All the assembly code is based on -O2 optimization level.

```
g++ -O2 -o func_inlining func_inlining.cpp
objdump -d func_inlining
```

inline

```
        bf4:       e8 e7 fe ff ff          callq  ae0 <gettimeofday@plt>
        bf9:       4c 8b 44 24 50          mov    0x50(%rsp),%r8
        bfe:       48 8b 7c 24 30          mov    0x30(%rsp),%rdi
        c03:       48 8d 0c ad 04 00 00    lea    0x4(,%rbp,4),%rcx
        c0a:       00
        c0b:       48 8b 74 24 70          mov    0x70(%rsp),%rsi
        c10:       31 c0                   xor    %eax,%eax
        c12:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
        c18:       8b 14 07                mov    (%rdi,%rax,1),%edx
        c1b:       41 03 14 00             add    (%r8,%rax,1),%edx
        c1f:       89 14 06                mov    %edx,(%rsi,%rax,1)
        c22:       48 83 c0 04             add    $0x4,%rax
        c26:       48 39 c8                cmp    %rcx,%rax
        c29:       75 ed                   jne    c18 <main+0xf8>
        c2b:       48 89 df                mov    %rbx,%rdi
        c2e:       31 f6                   xor    %esi,%esi
        c30:       31 db                   xor    %ebx,%ebx
        c32:       e8 a9 fe ff ff          callq  ae0 <gettimeofday@plt>
```

not inline

```
bf4:       e8 e7 fe ff ff          callq  ae0 <gettimeofday@plt>
bf9:       4c 8b 54 24 50          mov    0x50(%rsp),%r10
bfe:       4c 8b 4c 24 30          mov    0x30(%rsp),%r9
c03:       48 8d 0c 9d 04 00 00    lea    0x4(,%rbx,4),%rcx
c0a:       00
c0b:       4c 8b 44 24 70          mov    0x70(%rsp),%r8
c10:       31 d2                   xor    %edx,%edx
c12:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
c18:       41 8b 34 12             mov    (%r10,%rdx,1),%esi
c1c:       41 8b 3c 11             mov    (%r9,%rdx,1),%edi
c20:       e8 9b 03 00 00          callq  fc0 <_Z3addii>
c25:       41 89 04 10             mov    %eax,(%r8,%rdx,1)
c29:       48 83 c2 04             add    $0x4,%rdx
c2d:       48 39 ca                cmp    %rcx,%rdx
c30:       75 e6                   jne    c18 <main+0xf8>
c32:       31 f6                   xor    %esi,%esi
c34:       48 89 ef                mov    %rbp,%rdi
c37:       e8 a4 fe ff ff          callq  ae0 <gettimeofday@plt>
```

There are minor difference between inline and no-inline assembly code. We can see that at line c20 of no-inline, it calls the add() function rather than directly using addition.

(c)

Yes. My measured performance results match my expectations. Using inline is much faster than not using that. Because inline does not require function calling overhead, save overhead of variables push/pop on the stack, while function calling, and also save overhead of return call from a function.

Resource: Inline function Advantages, Disadvantage - C++ Articles

(d)

- Assembly code

  Below is the assembly code of original code, which is same as adding `always_inline` attribute.

```
      bf4:      e8 e7 fe ff ff          callq  ae0 <gettimeofday@plt>
      bf9:      4c 8b 44 24 50          mov    0x50(%rsp),%r8
      bfe:      48 8b 7c 24 30          mov    0x30(%rsp),%rdi
      c03:      48 8d 0c ad 04 00 00    lea    0x4(,%rbp,4),%rcx
      c0a:      00
      c0b:      48 8b 74 24 70          mov    0x70(%rsp),%rsi
      c10:      31 c0                   xor    %eax,%eax
      c12:      66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
      c18:      8b 14 07                mov    (%rdi,%rax,1),%edx
      c1b:      41 03 14 00             add    (%r8,%rax,1),%edx
      c1f:      89 14 06                mov    %edx,(%rsi,%rax,1)
      c22:      48 83 c0 04             add    $0x4,%rax
      c26:      48 39 c8                cmp    %rcx,%rax
      c29:      75 ed                   jne    c18 <main+0xf8>
      c2b:      48 89 df                mov    %rbx,%rdi
      c2e:      31 f6                   xor    %esi,%esi
      c30:      31 db                   xor    %ebx,%ebx
      c32:      e8 a9 fe ff ff          callq  ae0 <gettimeofday@plt>
```

- Performance

  As for the performance, which is also roughly the same. We can ensure that the compiler is in-lining the add() function by default.

	1	2	3	4	5	average
inline	96.376	97.325	101.906	98.325	100.958	98.978
not inline	188.414	189.119	187.68	188.495	186.244	187.99
original	95.204	96.829	96.874	95.656	104.46	97.8046

4. Loop transformations

- 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;
          }
      }
}
```

Step 1. Loop Invariant Hoisting

Pull non-loop-dependent calculations out of 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;
          }
      }
}
```

Step 2. 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++) {
      for (i=0; i<4; i++) {
          if (threshold < 4) {
              sum = sum + a[j][i];
          } else {
              sum = sum + a[j][i] + 1;
          }
      }
}
```

Step 3. Loop Unswitching

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;
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;
          }
      }
}
```

Step 4. Loop Unroll and Jam

Partially unroll one or more loops higher in the loop nest than the innermost loop, and then fuse (jam) resulting loops back together.

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

Step 5. Loop Reversal

Reverse the order of the loop iteration.

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

Step 6. Loop Peeling + Unrolling

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;
if (threshold < 4) {
      if (N > 0) {
          sum = sum + a[0][3];
          sum = sum + a[0][2];
          sum = sum + a[0][1];
          sum = sum + a[0][0];
      }
      for (j=1; j<N; j+=2) {
```

```
            for (i=3; i>=0; i--) {
                sum = sum + a[j][i];
                sum = sum + a[j+1][i];
            }
        }
} else {
        if (N > 0) {
            sum = sum + a[0][3] + 1;
            sum = sum + a[0][2] + 1;
            sum = sum + a[0][1] + 1;
            sum = sum + a[0][0] + 1;
        }
        for (j=1; j<N; j+=2) {
            for (i=3; i>=0; i--) {
                sum = sum + a[j][i] + 1;
                sum = sum + a[j+1][i] + 1;
            }
        }
}
```

5. Loop transformations.

(a) Loop fusion

```
//Original code
for (i=1; i<8; i++) {
        S1: a[i] = b[i] + 2;
        S2: c[i] = a[i] + b[i];
}
for (i=1; i<8; i++) {
        S3: a[i+1] = d[i] + 4;
}
//Code after Loop Interchange
for (i=1; i<8; i++) {
        S1: a[i] = b[i] + 2;
        S2: c[i] = a[i] + b[i];
```

```
        S3: a[i+1] = d[i] + 4;
}
```

- Loop Fusion is safe iff no data dependence between the nests becomes loop-carried data dependence of a different type (i.e. reverse the dep.)
  - Change from true dep. between A, B to anti-dep. between A, B
  - Or change from anti dep. between A, B to true dep. between A, B

- It is not safe.

  Before transformation:

  the value of array a is determined by S3 in the second for-loop. S1[i] ➔ O S3[i-1].

  After transformation:

  i = 1, S1: a[1] = b[1] + 2;
  S2: c[1] = a[1] + b[1];
  S3: a[2] = d[1] + 4;
  i = 2, S1: a[2] = b[2] + 2;
  S2: c[2] = a[2] + b[2];
  S3: a[3] = d[2] + 4;

  which is clear that S3[i] ➔ O S1[i+1] (loop-carried), the value of array a is modified and determined by S1.

(b) Loop interchange

```
//Original code
for (i=1; i<=4; i++) {
        for (j=1; j<=4; j++) {
            S1: a[i][j] = a[i+1][j-1];
        }
}
//Code after Loop Interchange
for (j=1; j<=4; j++) {
        for (i=1; i<=4; i++) {
            S1: a[i][j] = a[i+1][j-1];
        }
}
```

- 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').

- It is not safe.

  Before transformation:

  i = 1, j = 1, S1: a[1][1] = a[2][0];

  i = 1, j = 2, S1: a[1][2] = a[2][1];

  … …

  i = 2, j = 1, S1: a[2][1] = a[3][0];

  the value of a[1][2] equals to the original value of a[2][1], and we modifies a[2][1] after that.

  After transformation:

  j = 1, i = 1, S1: a[1][1] = a[2][0];

  j = 1, i = 2, S1: a[2][1] = a[3][0];

  … …

  j = 2, i = 1, S1: a[1][2] = a[2][1];

  the value of a[1][2] equals to the modified value of a[2][1], which equals to that of a[3][0],

  the execution order modified the final output.

(c) Loop fission

```
//Original code
for (i=1; i<=4; i++) {
      S1: a[i+1] = b[i] + c[i];
      S2: d[i] = a[i-1] + 4;
}
//Code after Loop Fission
for (i=1; i<=4; i++) {
      S1: a[i+1] = b[i] + c[i];
}
for (i=1; i<=4; i++) {
      S2: d[i] = a[i-1] + 4;
}
```

- Loop Fission is safe iff

- Statements involved in a cycle of loop-carried data dependences remain in the same loop AND
- If there exists a data dependence between 2 statements placed in different loops, the dependence type must not change

- It is safe.

  Before transformation:

  i = 1, S1: a[2] = b[1] + c[1];
  S2: d[1] = a[0] + 4;
  i = 2, S1: a[3] = b[2] + c[2];
  S2: d[2] = a[1] + 4;
  i = 3, S1: a[4] = b[3] + c[3];
  S2: d[3] = a[2] + 4;

  we have S1[i] ➜ T S2[i+2] (loop-carried).

  After transformation:

  loop1:
  i = 1, S1: a[2] = b[1] + c[1];
  i = 2, S1: a[3] = b[2] + c[2];
  i = 3, S1: a[4] = b[3] + c[3];
  loop2:
  i = 1, S2: d[1] = a[0] + 4;
  i = 2, S2: d[2] = a[1] + 4;
  i = 3, S2: d[3] = a[2] + 4;
  we have S1[i] ➜ T S2[i+2], too.

#Duke/courses/ece565