Code Optimization: Assignment N°5 Report

Duy Le - Cédric Léonard, 6/12/2021

Introduction

The assignment is about understanding the complier-generated assembly language

Question N°1

a. Compile the code with optimization level -O2 and generate assembly language code for the saxpy procedure:

The assembly code is generated by the following command: `gcc -c -g -O2 saxpy.c -std=c99` `objdump -d -S saxpy.o`

Here is the assembly code generated:

```
0000000000000000 <saxpy>:
/*
  Single-precision A*X plus Y, Y = alpha*X+Y
*/

void saxpy(int n, float alpha, float *X, float *Y) {
  for (int i=0; i<n; i++)
   0:   31 c0                   xor    %eax,%eax
   2:   85 ff                   test   %edi,%edi
   4:   7e 25                   jle    2b <saxpy+0x2b>
   6:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
   d:   00 00 00
    Y[i] = alpha*X[i] + Y[i];
  10:   f3 0f 10 0c 86          movss  (%rsi,%rax,4),%xmm1
  15:   f3 0f 59 c8             mulss  %xmm0,%xmm1
  19:   f3 0f 58 0c 82          addss  (%rdx,%rax,4),%xmm1
  1e:   f3 0f 11 0c 82          movss  %xmm1,(%rdx,%rax,4)
  23:   48 83 c0 01             add    $0x1,%rax
  for (int i=0; i<n; i++)
  27:   39 c7                   cmp    %eax,%edi
  29:   7f e5                   jg     10 <saxpy+0x10>
  2b:   f3 c3                   repz retq
}
```

- How are the parameters passed to the procedure The parameters are passed to the procedure as follow: n to register EDI, alpha to RDI, *X to RSI, *Y to RDX
- How is the loop implemented?

```
  for (int i=0; i<n; i++)
   0:   31 c0                    xor    %eax,%eax                # Set
i = 0
   2:   85 ff                    test   %edi,%edi                # if
n = 0
   4:   7e 25                    jle    2b <saxpy+0x2b>          #
then jump to 2b: terminate the program
    # Execute other code.
   23:   48 83 c0 01             add    $0x1,%rax                   #
Increase i

  for (int i=0; i<n; i++)
   27:   39 c7                    cmp    %eax,%edi               #
Compare i and n
   29:   7f e5                    jg     10 <saxpy+0x10>         #
Jump to the arithmetic operation if the the result is n>i
   2b:   f3 c3                    repz retq                      #
Otherwise, return
```

- How are the arithmetic operations done?

```
Y[i] = alpha*X[i] + Y[i];
  10:   f3 0f 10 0c 86          movss  (%rsi,%rax,4),%xmm1      `#
xmm1 = alpha
  15:   f3 0f 59 c8             mulss  %xmm0,%xmm1               #
xmm1 = x[i] x alpha
  19:   f3 0f 58 0c 82          addss  (%rdx,%rax,4),%xmm1       #
xmm1 += Y[i]
  1e:   f3 0f 11 0c 82          movss  %xmm1,(%rdx,%rax,4)       #
Y[i] = xmm1
```

- Instructions that have no effect, i.e. that are inserted by the compiler to align branch targets? `6:   66 2e 0f 1f 84 00 00    nopw %cs:0x0(%rax,%rax,1) # Used to align`

b. Compile the saxpy procedure with full optimizations (-O3) and look at the generated assembly language code.

The assembly code is generated by the following command: `gcc -c -g saxpy.c -o saxpyo3.o -O3 --std=c99` `objdump -d -S saxpyo3.o`

Here is the assembly code generated:

```
saxpyo3.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <saxpy>:
/*
  Single-precision A*X plus Y, Y = alpha*X+Y
*/

void saxpy(int n, float alpha, float *X, float *Y) {
  for (int i=0; i<n; i++)
   0:   85 ff                    test   %edi,%edi
   2:   0f 8e fb 00 00 00        jle    103 <saxpy+0x103>
   8:   48 8d 42 10              lea    0x10(%rdx),%rax
   c:   48 39 c6                 cmp    %rax,%rsi
   f:   48 8d 46 10              lea    0x10(%rsi),%rax
  13:   0f 93 c1                 setae  %cl
  16:   48 39 c2                 cmp    %rax,%rdx
  19:   0f 93 c0                 setae  %al
  1c:   08 c1                    or     %al,%cl
  1e:   0f 84 bc 00 00 00        je     e0 <saxpy+0xe0>
  24:   83 ff 05                 cmp    $0x5,%edi
  27:   0f 86 b3 00 00 00        jbe    e0 <saxpy+0xe0>
  2d:   0f 28 e0                 movaps %xmm0,%xmm4
  30:   41 89 f9                 mov    %edi,%r9d
  33:   31 c0                    xor    %eax,%eax
  35:   41 c1 e9 02              shr    $0x2,%r9d
    Y[i] = alpha*X[i] + Y[i];
  39:   0f 57 db                 xorps  %xmm3,%xmm3
  3c:   0f c6 e4 00              shufps $0x0,%xmm4,%xmm4
  40:   46 8d 04 8d 00 00 00     lea    0x0(,%r9,4),%r8d
  47:   00
  for (int i=0; i<n; i++)
  48:   31 c9                    xor    %ecx,%ecx
    Y[i] = alpha*X[i] + Y[i];
  4a:   0f 28 cb                 movaps %xmm3,%xmm1
  4d:   0f 28 d3                 movaps %xmm3,%xmm2
  50:   83 c1 01                 add    $0x1,%ecx
  53:   0f 12 0c 06              movlps (%rsi,%rax,1),%xmm1
```

```
57:   0f 12 14 02             movlps (%rdx,%rax,1),%xmm2
5b:   0f 16 4c 06 08          movhps 0x8(%rsi,%rax,1),%xmm1
60:   0f 16 54 02 08          movhps 0x8(%rdx,%rax,1),%xmm2
65:   0f 59 cc                mulps  %xmm4,%xmm1
68:   0f 58 ca                addps  %xmm2,%xmm1
6b:   0f 13 0c 02             movlps %xmm1,(%rdx,%rax,1)
6f:   0f 17 4c 02 08          movhps %xmm1,0x8(%rdx,%rax,1)
74:   48 83 c0 10             add    $0x10,%rax
78:   44 39 c9                cmp    %r9d,%ecx
7b:   72 cd                   jb     4a <saxpy+0x4a>
7d:   44 39 c7                cmp    %r8d,%edi
80:   0f 84 7d 00 00 00       je     103 <saxpy+0x103>
86:   49 63 c8                movslq %r8d,%rcx
89:   f3 0f 10 0c 8e          movss  (%rsi,%rcx,4),%xmm1
8e:   48 8d 04 8a             lea    (%rdx,%rcx,4),%rax
for (int i=0; i<n; i++)
92:   41 8d 48 01             lea    0x1(%r8),%ecx
  Y[i] = alpha*X[i] + Y[i];
96:   f3 0f 59 c8             mulss  %xmm0,%xmm1
for (int i=0; i<n; i++)
9a:   39 cf                   cmp    %ecx,%edi
  Y[i] = alpha*X[i] + Y[i];
9c:   f3 0f 58 08             addss  (%rax),%xmm1
a0:   f3 0f 11 08             movss  %xmm1,(%rax)
for (int i=0; i<n; i++)
a4:   7e 5d                   jle    103 <saxpy+0x103>
  Y[i] = alpha*X[i] + Y[i];
a6:   48 63 c9                movslq %ecx,%rcx
for (int i=0; i<n; i++)
a9:   41 83 c0 02             add    $0x2,%r8d
  Y[i] = alpha*X[i] + Y[i];
ad:   f3 0f 10 0c 8e          movss  (%rsi,%rcx,4),%xmm1
b2:   48 8d 04 8a             lea    (%rdx,%rcx,4),%rax
for (int i=0; i<n; i++)
b6:   44 39 c7                cmp    %r8d,%edi
  Y[i] = alpha*X[i] + Y[i];
b9:   f3 0f 59 c8             mulss  %xmm0,%xmm1
bd:   f3 0f 58 08             addss  (%rax),%xmm1
c1:   f3 0f 11 08             movss  %xmm1,(%rax)
for (int i=0; i<n; i++)
c5:   7e 41                   jle    108 <saxpy+0x108>
  Y[i] = alpha*X[i] + Y[i];
c7:   4d 63 c0                movslq %r8d,%r8
ca:   f3 42 0f 59 04 86       mulss  (%rsi,%r8,4),%xmm0
d0:   4a 8d 04 82             lea    (%rdx,%r8,4),%rax
```

```
  d4:   f3 0f 58 00             addss  (%rax),%xmm0
  d8:   f3 0f 11 00             movss  %xmm0,(%rax)
  dc:   c3                      retq
  dd:   0f 1f 00                nopl   (%rax)
  for (int i=0; i<n; i++)
  e0:   31 c0                   xor    %eax,%eax
  e2:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    Y[i] = alpha*X[i] + Y[i];
  e8:   f3 0f 10 0c 86          movss  (%rsi,%rax,4),%xmm1
  ed:   f3 0f 59 c8             mulss  %xmm0,%xmm1
  f1:   f3 0f 58 0c 82          addss  (%rdx,%rax,4),%xmm1
  f6:   f3 0f 11 0c 82          movss  %xmm1,(%rdx,%rax,4)
  fb:   48 83 c0 01             add    $0x1,%rax
  for (int i=0; i<n; i++)
  ff:   39 c7                   cmp    %eax,%edi
 101:   7f e5                   jg     e8 <saxpy+0xe8>
 103:   f3 c3                   repz retq
 105:   0f 1f 00                nopl   (%rax)
 108:   f3 c3                   repz retq
```

The O3 optimization level, which is the highest one, performs the serious optimizing operation. In the above assembly code, the loop has been unrolled, in addition to that, function inlining and automatic vectorization have been performed.

c. Compile the saxpy procedure with -O2 optimization and the additional flag -mfpmath=387

The assembly code are generated by the following command: `gcc -c -g saxpy.c -o saxpyo2mfpm.o -O2 -mfpmath=387 --std=c99` `objdump -d -S saxpyo2mfpm.o`

Here is the assembly code generated:

```
saxpyo2mfpm.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <saxpy>:
/*
  Single-precision A*X plus Y, Y = alpha*X+Y
*/
```

```
void saxpy(int n, float alpha, float *X, float *Y) {
  for (int i=0; i<n; i++)
   0:   85 ff                   test   %edi,%edi
   2:   7e 21                   jle    25 <saxpy+0x25>
   4:   f3 0f 11 44 24 f0       movss  %xmm0,-0x10(%rsp)
   a:   31 c0                   xor    %eax,%eax
   c:   d9 44 24 f0             flds   -0x10(%rsp)
    Y[i] = alpha*X[i] + Y[i];
  10:   d9 c0                   fld    %st(0)
  12:   d8 0c 86                fmuls  (%rsi,%rax,4)
  15:   d8 04 82                fadds  (%rdx,%rax,4)
  18:   d9 1c 82                fstps  (%rdx,%rax,4)
  1b:   48 83 c0 01             add    $0x1,%rax
  for (int i=0; i<n; i++)
  1f:   39 c7                   cmp    %eax,%edi
  21:   7f ed                   jg     10 <saxpy+0x10>
  23:   dd d8                   fstp   %st(0)
  25:   f3 c3                   repz retq
```

The loop structure is similar to the previous case in question a. However, the mathematical operation is different from the above. With -mfpmath=387, the assembly code produced to look much cleaner and seem to be more efficient.

Question N°2

Compile the function med3 with optimization level -O2 and generate assembly code for it. Analyze and explain the assembly code like in question 1a

The assembly code are generated by the following command: `gcc -c -g med3.c -o med3o2.o -O2 --std=c99` `objdump -d -S med3o2.o`

Here is the assembly code generated:

```
med3o2.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <med3>:

int med3(int a, int b, int c) {
  int mid;
  if ( a > b )   {
```

```
   0:   39 f7                   cmp    %esi,%edi
int med3(int a, int b, int c) {
   2:   89 f0                   mov    %esi,%eax
  if ( a > b )   {
   4:   7e 12                   jle    18 <med3+0x18>
    if ( c > b) {
   6:   39 d6                   cmp    %edx,%esi
   8:   7d 19                   jge    23 <med3+0x23>
   a:   39 d7                   cmp    %edx,%edi
   c:   89 d0                   mov    %edx,%eax
   e:   0f 4e c7                cmovle %edi,%eax
  11:   c3                      retq
  12:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    else {
      mid = b;
    }
  }
  else {
    if ( b > c ) {
  18:   39 d6                   cmp    %edx,%esi
  1a:   7e 07                   jle    23 <med3+0x23>
  1c:   39 d7                   cmp    %edx,%edi
  1e:   89 d0                   mov    %edx,%eax
  20:   0f 4d c7                cmovge %edi,%eax
    else {
      mid = b;
    }
  }
  return mid ;
}
  23:   f3 c3                   repz retq
```

• how are the parameters passed to the procedure? The first parameter (a) is passed into register EDI and the second (b) in ESI, the third parameter (c) is passed into register EDX

• how is the condition block implemented? • how are the arithmetic operations done?

```
med3o2.o:     file format elf64-x86-64

Disassembly of section .text:
```

```
0000000000000000 <med3>:

int med3(int a, int b, int c) {
  int mid;
  if ( a > b )   {
   0:   39 f7                   cmp    %esi,%edi               #
compare b with a
int med3(int a, int b, int c) {
   2:   89 f0                   mov    %esi,%eax               #
move esi (b) to eax
  if ( a > b )   {
   4:   7e 12                   jle    18 <med3+0x18>          # if
less than or equal, jump to 18
    if ( c > b) {
   6:   39 d6                   cmp    %edx,%esi               #
compare c with b
   8:   7d 19                   jge    23 <med3+0x23>          # if
greater than or equal
   a:   39 d7                   cmp    %edx,%edi               #
compare c with a
   c:   89 d0                   mov    %edx,%eax               #
move edx (c) to eax
   e:   0f 4e c7                cmovle %edi,%eax               #
move edi (a) to eax if less than or equal
  11:   c3                      retq                           #
return
  12:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)        # has
no effect, insert by the compiler to align branch targets
      mid = b;
    }
  }
  else {
    if ( b > c ) {
  18:   39 d6                   cmp    %edx,%esi               #
compare c to b
  1a:   7e 07                   jle    23 <med3+0x23>          # if
less than or equal, jump to 23 (return)
  1c:   39 d7                   cmp    %edx,%edi               #
compare c to a
  1e:   89 d0                   mov    %edx,%eax               #
move c to eax
  20:   0f 4d c7                cmovge %edi,%eax               #
move a to eax if greater than or equal
    else {
      mid = b;
```

```
    }
  }
  return mid ;
}
  23:   f3 c3                   repz retq                       #
return
```

• can you recognize any instructions that have no effect, i.e. that are inserted by the compiler to align branch targets? `assembly`
`12:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)         # has no effect, insert by the compiler to align branch targets`

Question N°3

The procedure in the attached file call_max.c illustrates how procedure calls are implemented in the generated assembly language

a. The assembly code are generated by the following command: `gcc -c -g call_max.c -o call_max-fno-inline.o -O2 -fno-inline --std=c99` `objdump -d -S call_max-fno-inline.o`

Here is the assembly code generated with explanations:

```
call_max-fno-inline.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <max>:
int max(int a, int b) {
/*
The first parameter (a) is passed into register EDI and the second
(b) in ESI
*/
   0: 39 f7                 cmp    %esi,%edi
# compare b to a
   2: 89 f0                 mov    %esi,%eax
# move b to eax
   4: 0f 4d c7              cmovge %edi,%eax
# move a to eax if greater than or equal
  if (a>b) return a;
  else return b;
}
   7: c3                    retq
```

```
# return
   8: 0f 1f 84 00 00 00 00  nopl   0x0(%rax,%rax,1)
# has no effect
   f: 00

0000000000000010 <select>:

void select (int *a, int *b, int len) {
/*
The first parameter (*a) is passed into register EDI and the second
(*b) in ESI, the third parameter (len) is passed into register EDX
*/
  for (int i=0; i<len; i++) {
  10: 85 d2                test   %edx,%edx
# if len = 0
  12: 7e 3c                jle    50 <select+0x40>
# then return by jumping into 50
void select (int *a, int *b, int len) {
  14: 41 55                push   %r13
# push r13
  16: 8d 42 ff             lea    -0x1(%rdx),%eax
# eax = len - 1
  19: 41 54                push   %r12
# push r12
  1b: 4c 8d 2c 85 04 00 00  lea    0x4(,%rax,4),%r13
# r13 = r12 + 4
  22: 00
  23: 49 89 f4             mov    %rsi,%r12
# move rsi to r12
  26: 55                   push   %rbp
# push register base pointer
  27: 48 89 fd             mov    %rdi,%rbp
# move rdi to rbp
  2a: 53                   push   %rbx
# push rbx
  for (int i=0; i<len; i++) {
  2b: 31 db                xor    %ebx,%ebx
# i = 0
  2d: 0f 1f 00             nopl   (%rax)
# no effect
    a[i] = max(a[i], b[i]);
  30: 41 8b 34 1c          mov    (%r12,%rbx,1),%esi
# move r12 to esi
  34: 8b 7c 1d 00          mov    0x0(%rbp,%rbx,1),%edi
# move rbp to edi
```

```
  38: e8 00 00 00 00        callq  3d <select+0x2d>
# calq 3d
  3d: 89 44 1d 00           mov    %eax,0x0(%rbp,%rbx,1)
# move eax to rbp
  41: 48 83 c3 04           add    $0x4,%rbx
# rbx += 4
  for (int i=0; i<len; i++) {
  45: 4c 39 eb              cmp    %r13,%rbx
# compare r13 with rbx
  48: 75 e6                 jne    30 <select+0x20>
# if not equal, jump to 30
  }


}
  4a: 5b                    pop    %rbx
# pop rbx
  4b: 5d                    pop    %rbp
# pop rbp
  4c: 41 5c                 pop    %r12
# pop r12
  4e: 41 5d                 pop    %r13
# pop r13
  50: f3 c3                 repz retq
# return
```

b. The assembly code are generated by the following command: `gcc -c -g call_max.c -o call_max.o -O2 --std=c99` `objdump -d -S call_max.o`

Here is the assembly code generated:

```
call_max-fno-inline.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <max>:
int max(int a, int b) {
/*
The first parameter (a) is passed into register EDI and the second
(b) in ESI
*/
   0:   39 f7                   cmp    %esi,%edi               #
compare b to a
```

```
   2:   89 f0                   mov    %esi,%eax               #
move b to eax
   4:   0f 4d c7                cmovge %edi,%eax               #
move a to eax if greater than or equal
  if (a>b) return a;
  else return b;
}
   7:   c3                      retq                           #
return
   8:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)        # no
effect
   f:   00

0000000000000010 <select>:

void select (int *a, int *b, int len) {
/*
The first parameter (*a) is passed into register EDI and the second
(*b) in ESI, the third parameter (len) is passed into register EDX
*/
  for (int i=0; i<len; i++) {
  10:   31 c0                   xor    %eax,%eax               # set
i = 0
  12:   85 d2                   test   %edx,%edx               # if
len = 0
  14:   7e 1f                   jle    35 <select+0x25>        #
then return by jumping to 35
  16:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)    # no
effect, used to align
  1d:   00 00 00
  20:   8b 0c 86                mov    (%rsi,%rax,4),%ecx      #
move rsi to ecx
  23:   39 0c 87                cmp    %ecx,(%rdi,%rax,4)      #
compare ecx to rdi
  26:   0f 4d 0c 87             cmovge (%rdi,%rax,4),%ecx      #
move rdi to ecx if greater than or equal
    a[i] = max(a[i], b[i]);
  2a:   89 0c 87                mov    %ecx,(%rdi,%rax,4)      #
move ecx to rdi
  2d:   48 83 c0 01             add    $0x1,%rax               # i++
  for (int i=0; i<len; i++) {
  31:   39 c2                   cmp    %eax,%edx               #
compare i to len
  33:   7f eb                   jg     20 <select+0x10>        #
jump to 20 if greater than
```

```
  35:   f3 c3                   repz retq                       #
return
```

With function inlining enabled, the code is more simple and uses fewer registers to perform.