上机作业 A4：循环向量化

布置周： 11 月 8 日
提交周： 11 月 22 日

目标：通过本次作业，希望同学们进一步掌握 Intel 的向量化指令以及编译器是如何对循环进行向量化优化的，并通过理解和分析编译器生成的汇编代码，确认编译器进行向量化优化的效果以及如何对源程序或者编译选项进行一些小修改从而让编译器能够生成更加高质量的向量化代码。

作业要求：用 clang 开源编译器对 https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/assignments/MIT6_172F18_hw3.zip 中的多个程序示例进行循环向量化相关的实验，理解影响循环向量化的一些关键因素，掌握相关的循环向量化的优化实践，并且通过对向量化后程序的运行性能测量与分析，从而对循环向量化带来的具体性能提升获得一个感性的认识。

*备注：此作业从 MIT 6.172 Performance Engineering of Software Systems "Homework 3: Vectorization" 改编而来（源文档地址：https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/assignments/MIT6_172F18hw3.pdf），MIT6_172F18hw3.pdf 可以作为参考资料，但**请同学们按照本改编后的作业要求完成作业**。*

1. 循环向量化优化

recitation3/example1.c 源文件如下图所示：.

```c
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * a,  uint8_t * b) {
  uint64_t i;

  for (i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example1.o
```

命令运行的输出结果如下：

```
bhuang@LAPTOP-BHUANG:~/Courses/A4/recitation3$ make clean; make ASSEMBLE=1 VECTORIZE=1 example1.o
rm -rf
rm -f *.o *.s .cflags perf.data *.lst
clang -Wall -g -std=gnu99 -O3 -Rpass=loop-vectorize -Rpass-missed=loop-vectorize  -S  -c example1.
c
example1.c:12:3: remark: vectorized loop (vectorization width: 16, interleaved count: 2) [-Rpass=l
oop-vectorize]
  for (i = 0; i < SIZE; i++) {
  ^
```

Write-up 1：请同学们对照着 example1.c 仔细理解上一命令的输出文件 example1.s，画出汇编代码的控制流图。同时比较一下你得到的 example1.s 跟下图中另一版本的编译器生成的 example1.s 有什么区别。详细说明你得到的 example1.s 中.LBB0_2 内向量化后代码的循环控制逻辑跟下图中.LBB0_2 内向量化后代码的循环控制逻辑有何不同，阐述你对这两种代码生成方法各自优缺点的理解。(请同学们注明一下自己实验环境下的 clang 版本)

```
# %bb.0:                                # %entry
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:a <- %rdi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:i <- 0
        .loc    1 12 3 prologue_end     # example1.c:12:3

        leaq    65536(%rsi), %rax
        cmpq    %rdi, %rax
        jbe     .LBB0_2
# %bb.1:                                # %entry
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        leaq    65536(%rdi), %rax
        cmpq    %rsi, %rax
        jbe     .LBB0_2
# %bb.4:                                # %for.body.preheader
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
        .loc    1 0 3 is_stmt 0         # example1.c:0:3

        movq    $-65536, %rax           # imm = 0xFFFF0000

        .p2align        4, 0x90
.LBB0_5:                                # %for.body
                                        # =>This Inner Loop Header: Depth=1
        #DEBUG_VALUE: test:b <- %rsi
        #DEBUG_VALUE: test:a <- %rdi
.Ltmp0:
        .loc    1 13 13 is_stmt 1       # example1.c:13:13

        movzbl    65536(%rsi,%rax), %ecx
        .loc    1 13 10 is_stmt 0       # example1.c:13:10

        addb    %cl, 65536(%rdi,%rax)
        .loc    1 13 13                 # example1.c:13:13

        movzbl    65537(%rsi,%rax), %ecx
        .loc    1 13 10                 # example1.c:13:10

        addb    %cl, 65537(%rdi,%rax)
        .loc    1 13 13                 # example1.c:13:13

        movzbl    65538(%rsi,%rax), %ecx
        .loc    1 13 10                 # example1.c:13:10

        addb    %cl, 65538(%rdi,%rax)
        .loc    1 13 13                 # example1.c:13:13

        movzbl    65539(%rsi,%rax), %ecx
        .loc    1 13 10                 # example1.c:13:10

        addb    %cl, 65539(%rdi,%rax)
```

```
.Ltmp1:
    .loc    1 12 17 is_stmt 1       # example1.c:12:17

    addq    $4, %rax
.Ltmp2:
    .loc    1 12 3 is_stmt 0        # example1.c:12:3

    jne .LBB0_5
    jmp .LBB0_6
.LBB0_2:                                # %vector.body.preheader
    #DEBUG_VALUE: test:b <- %rsi
    #DEBUG_VALUE: test:a <- %rdi
    .loc    1 0 3                   # example1.c:0:3

    movq    $-65536, %rax           # imm = 0xFFFF0000
    .p2align    4, 0x90
.LBB0_3:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
    #DEBUG_VALUE: test:b <- %rsi
    #DEBUG_VALUE: test:a <- %rdi
.Ltmp3:
    .loc    1 13 13 is_stmt 1       # example1.c:13:13

    movdqu  65536(%rsi,%rax), %xmm0
    movdqu  65552(%rsi,%rax), %xmm1
    .loc    1 13 10 is_stmt 0       # example1.c:13:10

    movdqu  65536(%rdi,%rax), %xmm2
    paddb   %xmm0, %xmm2
    movdqu  65552(%rdi,%rax), %xmm0
    movdqu  65568(%rdi,%rax), %xmm3
    movdqu  65584(%rdi,%rax), %xmm4
    movdqu  %xmm2, 65536(%rdi,%rax)
    paddb   %xmm1, %xmm0
    movdqu  %xmm0, 65552(%rdi,%rax)
    .loc    1 13 13                 # example1.c:13:13

    movdqu  65568(%rsi,%rax), %xmm0
    .loc    1 13 10                 # example1.c:13:10

    paddb   %xmm3, %xmm0
    .loc    1 13 13                 # example1.c:13:13

    movdqu  65584(%rsi,%rax), %xmm1
    .loc    1 13 10                 # example1.c:13:10

    movdqu  %xmm0, 65568(%rdi,%rax)
    paddb   %xmm4, %xmm1
    movdqu  %xmm1, 65584(%rdi,%rax)
.Ltmp4:
    .loc    1 12 26 is_stmt 1       # example1.c:12:26

    addq    $64, %rax
    jne .LBB0_3
```

2. Alias 人为消除（using restrict attribute）

修改 example1.c，为 test 函数的两个参数 a 和 b 分别添加 restrict 属性，如下图所示：

```c
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * restrict a,  uint8_t * restrict b) {
  uint64_t i;

  for (i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example1.o
```

Write-up 2：请同学们仔细理解这个命令所产生的新的输出文件 example1.s 与 Write-up 1 中所分析的 example1.s 有何区别，简单阐述为何会有这个区别。

3. 内存对齐（Alignment）

在上一步修改的基础上进一步修改 example1.c，如下图所示：

```c
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * restrict a,  uint8_t * restrict b) {
  uint64_t i;

  a = __builtin_assume_aligned(a, 16);
  b = __builtin_assume_aligned(b, 16);

  for (i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example1.o
```

Write-up 3：请同学们仔细理解这个命令所产生的新的输出文件 example1.s 与 Write-up 2 中所分析的 example1.s 有何区别，简单阐述为何会有这个区别。

4. 生成 AVX2 代码

对上一步中的 example1.c 运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 AVX2=1 example1.o
```

Write-up 4：请同学们仔细理解这个命令所产生的新的输出文件 example1.s 与 Write-up 3 中所分析的 example1.s 有何区别，简单阐述为何会有这个区别。为使编译器生成对齐内存访问的 AVX2 指令，应该对 example1.c 再做何修改？请附上修改后的代码。

5. if 语句 vs. 问号表达式

recitation3/example2.c 如下图所示

```c
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * restrict a, uint8_t * restrict b) {
  uint64_t i;

  uint8_t * x = __builtin_assume_aligned(a, 16);
  uint8_t * y = __builtin_assume_aligned(b, 16);

  for (i = 0; i < SIZE; i++) {
    /* max() */
    if (y[i] > x[i]) x[i] = y[i];
  }
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example2.o
```

可以生成 example2.s，把 example2.s 重命名成 example2.s.ORG

改变 example2.c 中循环内的语句，改变后的程序如下图所示（即把 if 语句改成了语义等价的问号表达式）：

```c
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * restrict a, uint8_t * restrict b) {
  uint64_t i;

  uint8_t * x = __builtin_assume_aligned(a, 16);
  uint8_t * y = __builtin_assume_aligned(b, 16);

  for (i = 0; i < SIZE; i++) {
    /* max() */
    x[i] = (y[i] > x[i]) ? y[i] : x[i];
  }
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example2.o
```

可以生成新的 example2.s

Write-up 5: 请认真比较 example2.s 和 example2.s.ORG，尝试解释一下为什么新的 examplse2.s 含有比较高效的向量化代码。

6. 循环向量化一定最快吗？

recitation3/example3.c 如下图所示：

```c
// Copyright (c) 2015 MIT License by 6.172 Staff

#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#define SIZE (1L << 16)

void test(uint8_t * restrict a, uint8_t * restrict b) {
  uint64_t i;

  for (i = 0; i < SIZE; i++) {
    a[i] = b[i + 1];
  }
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example3.o
```

可以生成 example3.s。

Write-up 6: 请认真理解 example3.s，尝试解释一下为什么 examplse3.s 中没有出现向量化指令。编译器对这个循环进行向量化优化是否可以生成更加高性能的代码？为什么？

7. 浮点运算的顺序

recitation3/example4.c 中的 test 函数如下图所示：

```c
#define SIZE (1L << 16)

double test(double * restrict a) {
  size_t i;

  double *x = __builtin_assume_aligned(a, 16);

  double y = 0;

  for (i = 0; i < SIZE; i++) {
    y += x[i];
  }
  return y;
}
```

运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example4.o
```

可以生成 example4.s，把 example4.s 重命名成 example4.s.ORG。

修改 Makefile，把第 6 行的

```
    CFLAGS := -Wall -g -std=gnu99
```

改成

```
    CFLAGS := -Wall -g -std=gnu99 -ffast-math
```

再运行如下命令：

```
$make clean; make ASSEMBLE=1 VECTORIZE=1 example4.o
```

可以生成新的 example4.s。

Write-up 7: 请比较 example4.s.ORG 和 example4.s 中跟 test 函数对应的汇编代码，解释这两个版本中汇编代码的差别，并解释导致这个差别的原因？

8. 性能影响

从 recitation3 目录切换到 homework3 目录，做一个简单的实验去验证向量化优化带来的性能提升。

- 原始性能：
  $make; time ./loop
- 向量化后的性能：
  $make VECTORIZE=1; time ./loop
- 用 AVX2 指令优化后的性能：
  $make VECTORIZE=1 AVX2=1; time ./loop

Write-up 8：设计一个实验方案，最后用图示化的形式科学地比较上述三种情况下生成代码的性能，并尝试去解释性能提升的原因。请注意实验结果的描述需要包含对你的实验平台的描述，而且性能提升的解释也需要结合你的实验平台特点。

注：

【

对于使用 Mac（M1/M2）机器的同学，Writeup 1 - Writeup 7 可以在 Mac（M1/M2）上通过交叉编译（安装 libc6-dev-amd64-cross 并修改 Makefile）生成相应的汇编文件后做分析，Writeup 8 可以有三个选择

1） 在 Mac（M1/M2）上用 Qemu 仿真器运行 X86-64 的执行文件并记录性能

2） 在 Mac（M1/M2）上借助于 Rosetta 2 来运行生成的 X86-64 可执行文件并记录性能

3） 把编译出来的 binary 拿到同学的 X86-64 机器或者云实例上去测性能数据

】