Homework4: Vectorization

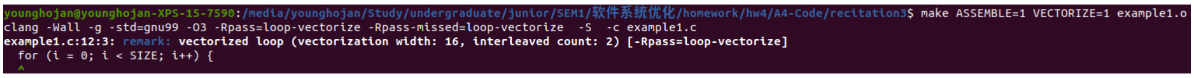
杨浩然 10195501441

操作系统： ubuntu 20.04

2021-10-12

Write-up 1

Q: Look at the assembly code. The compiler has translated the code to set the start index at -2^{16} and adds to it for each memory access. Why doesn't it set the start index to 0 and use small positive offsets?



A: 在我的系统下生成的汇编代码 example1.s 与题中的代码并不一致。

```
## 在本地生成的代码

# %bb.4:
	#DEBUG_VALUE: test:i <- 0
	#DEBUG_VALUE: test:b <- $rsi
	#DEBUG_VALUE: test:a <- $rdi
	.loc	1 0 3 is_stmt 0         # example1.c:0:3
	xorl	%eax, %eax              # 这里将 %eax 置为 0
```

```
# 题中的代码

# %bb.4:
	#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
```

可以看到，在我生成的 example1.s 中，`xorl %eax, %eax` 将 %eax 置 0；在题中代码的相同位置，`movq $-65536, %rax` 将 -65536 赋给 %eax。

再比如：

```
.LBB0_5:                                # =>This Inner Loop Header: Depth=1
	#DEBUG_VALUE: test:b <- $rsi
	#DEBUG_VALUE: test:a <- $rdi
	#DEBUG_VALUE: test:i <- $rax
	.loc	1 13 13 is_stmt 1       # example1.c:13:13
	movzbl	(%rsi,%rax), %ecx
	.loc	1 13 10 is_stmt 0       # example1.c:13:10
	addb	%cl, (%rdi,%rax)
	.loc	1 13 13                 # example1.c:13:13
	movzbl	1(%rsi,%rax), %ecx
	.loc	1 13 10                 # example1.c:13:10
```

```
    addb    %cl, 1(%rdi,%rax)
    .loc    1 13 13                 # example1.c:13:13
    movzbl  2(%rsi,%rax), %ecx
    .loc    1 13 10                 # example1.c:13:10
    addb    %cl, 2(%rdi,%rax)
    .loc    1 13 13                 # example1.c:13:13
    movzbl  3(%rsi,%rax), %ecx
    .loc    1 13 10                 # example1.c:13:10
    addb    %cl, 3(%rdi,%rax)
```

可以看出，这份代码 *start index to 0 and use small positive offsets*，与题意完全相反。

Write-up 2

Q: This code is still not aligned when using AVX2 registers. Fix the code to make sure it uses aligned moves for the best performance.

A: 根据 Intel ® C++ Compiler Classic Developer Guide and Reference（文件附后），AVX2 是 256 bits (32 bytes) 的指令集。

作如下修改:

```
  uint8_t * x = __builtin_assume_aligned(a, 32);
  uint8_t * y = __builtin_assume_aligned(b, 32);
```

Write-up 3

Q: Provide a theory for why the compiler is generating dramatically different assembly.

A: 两份代码的不同之处在于条件判断的方式:

"assembly does not vectorize nicely" 的是采用的 `if` 判断，如下:

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

"the vectorized assembly with the movdqa and pmaxub instructions" 的是采用的三目运算符，如下:

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

可能是对 if statement 和 ternary operator 的优化不同（ternary operator 没有逐元素比较两数组...？）。

Write-up 4

Q: Inspect the assembly and determine why the assembly does not include instructions with vector registers. Do you think it would be faster if it did vectorize? Explain.

A: 这里先用 `addq   $1, %rsi` 把 b 往后移，然后调 `memcpy` 把 b 复制给 a。由于内存没有对齐，所以没有关于向量寄存器的指令。如果做向量化的话，可以把 a 往后移动一个位置来与 b 重新对齐，采用向量化应该会更快。

Write-up 5

Q: Check the assembly and verify that it does in fact vectorize properly. Also what do you notice when you run the command `clang -O3 example4.c -o example4; ./example4` with and without the `-ffast-math` flag? Specifically, why do you a see a difference in the output.

A: The ffast-math flag was already set from a prior experiment! The addsd command is replaced by a addpd command.

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/recitation3$ clang -O3 example4.c -o example4; ./example4
The decimal floating point sum result is: 11.667578
The raw floating point sum result is: 0x1.755cccec10aa5p+3
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/recitation3$ clang -O3 -ffast-math example4.c -o example4; ./example4
The decimal floating point sum result is: 11.667578
The raw floating point sum result is: 0x1.755cccec10aa3p+3
```

使用 `%a` 输出的（p-计数法表示）sum 在最后一位上有差异。

Write-up 6

Q: What speedup does the vectorized code achieve over the unvectorized code? What additional speedup does using -mavx2 give? You may wish to run this experiment several times and take median elapsed times; you can report answers to the nearest 100% (e.g., 2 × , 3 × , etc). What can you infer about the bit width of the default vector registers on the awsrun machines? What about the bit width of the AVX2 vector registers? Hint: aside from speedup and the vectorization report, the most relevant information is that the data type for each array is uint32_t.

```
(base) younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/homework3$ make
clang -Wall -std=gnu99 -g -O3 -DNDEBUG -fno-vectorize  -c loop.c
clang -o loop loop.o -lrt
(base) younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/homework3$ ./loop
Elapsed execution time: 0.041920 sec; N: 1024, I: 100000, __OP__: +, __TYPE__: uint32_t
(base) younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/homework3$ make VECTORIZE=1
clang -Wall -std=gnu99 -g -O3 -DNDEBUG -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -ffast-math  -c loop.c
loop.c:70:9: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
        for (j = 0; j < N; j++) {
        ^
clang -o loop loop.o -lrt
(base) younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/homework3$ ./loop
Elapsed execution time: 0.012451 sec; N: 1024, I: 100000, __OP__: +, __TYPE__: uint32_t
(base) younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/homework/hw4/A4-Code/homework3$ make VECTORIZE=1 AVX2=1
clang -Wall -std=gnu99 -g -O3 -DNDEBUG -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -ffast-math -mavx2  -c loop.c
loop.c:70:9: remark: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
        for (j = 0; j < N; j++) {
        ^
clang -o loop loop.o -lrt
```

每种编译选项进行 5 次重复实验取中位数，结果如下:

- 不进行向量化:

  Elapsed execution time: **0.040384** sec

- 进行向量化:

  Elapsed execution time: **0.012676** sec

- 使用 AVX2 指令

  Elapsed execution time: **0.007966** sec

向量化相对于非向量化的 speedup 大概为 4x。SSE 向量寄存器通常是 128(4 x 32) 位宽，和 speedup 是吻合的。

使用 AVX2 指令集相对于不使用 AVX2 指令集的 speedup 大概为 2x。查查 intel 的文档，AVX2 向量寄存器是 256(128 x 2) 位宽，和 speedup 也是吻合的。