Assignment 4

Writeup-1: Look at the assembly code above. 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?

这里的size定义为1L<<16也就是2^16。向量化是重写一个循环的过程，它不是对数组中的单个元素进行 n 次处理，而是同时对数组中的2个元素进行 2^{16}/2次处理，所以不是从index 0开始的。

Write-up 2: This code is still not aligned when using AVX2 registers. Fix the code to make

sure it uses aligned moves for the best performance.

```
__builtin_assume_aligned(a, 16);
```

将这里的16改成32，因为AVX2 是 256 bits (也就是32 bytes) 的指令集。

Write-up 3: Provide a theory for why the compiler is generating dramatically different assembly.

__builtin_assume_aligned(a, 16)告诉编译器输入的C指针数据是16字节对齐的

MOVDQA - Move Aligned Double Quadword 将双四字从源操作数(第二个操作数)移动到目标操作数(第一个操作数)。

PMAXUB-Maximum of Packed Unsigned Byte Integers 对目标操作数(第一个操作数)和源操作数(第二个操作数)中打包的无符号字节整数执行 SIMD 比较，并将每对字节整数的最大值返回到目标操作数。

由于 c 内存模型不允许 Clang 向内存位置引入额外的写操作。如果其他线程在调用上面的例程时修改 a []，那么这些额外的写操作可能会影响修改最后的结果。这里使用了向量化的max指令，不需要使用任何显式的内部实现。最后汇编初的结果和之前也差距很大。

Write-up 4: 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.

因为指针没有对齐，所以没法使用向量化寄存器。使用向量化的结果会更快。

向量化意味着一条指令可以并行地对多个操作数执行相同的操作。

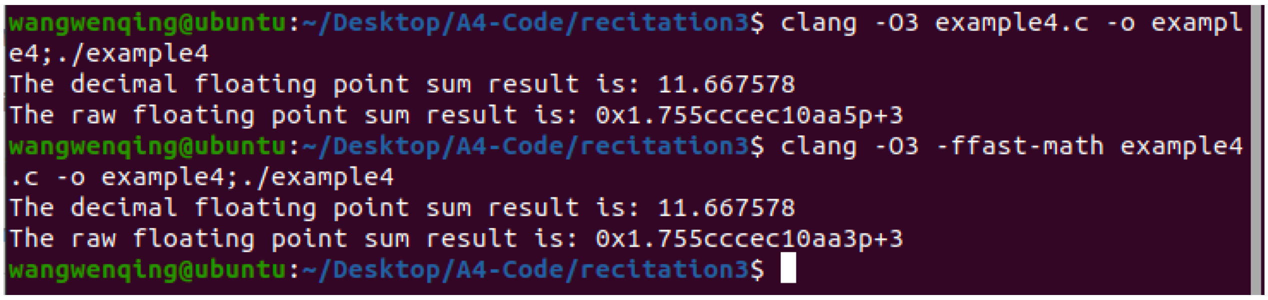
假设正在处理的32位，单精度实数，这意味着一条指令可以一次执行8个操作， 所以(至少在理论上)你可以完成 n 次赋值只用 n/8条乘法指令。至少在理论上，这应该允许操作以每次执行一条指令所允许的8倍的速度完成。

Write-up 5: 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 flflag? Specifiﬁcally, why do you a see a difference in the output.

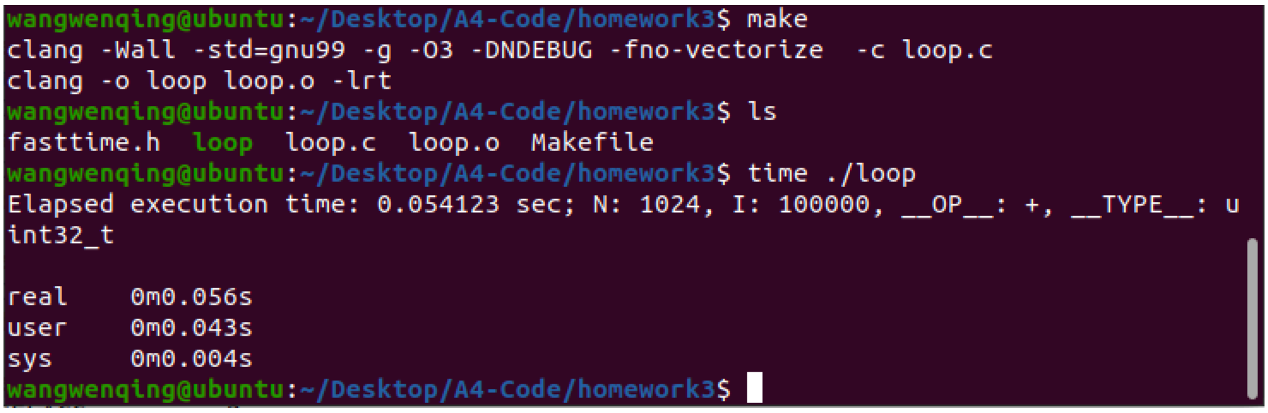


```
wangwenqing@ubuntu:~/Desktop/A4-Code/recitation3$ clang -O3 example4.c -o exampl
e4;./example4
The decimal floating point sum result is: 11.667578
The raw floating point sum result is: 0x1.755cccec10aa5p+3
wangwenqing@ubuntu:~/Desktop/A4-Code/recitation3$ clang -O3 -ffast-math example4
.c -o example4;./example4
The decimal floating point sum result is: 11.667578
The raw floating point sum result is: 0x1.755cccec10aa3p+3
wangwenqing@ubuntu:~/Desktop/A4-Code/recitation3$ 
```

查看汇编代码，已被正确向量化

当加了 -ffast-math 时，循环执行的顺序发生改变，也因此最后的sum result有微小的差距

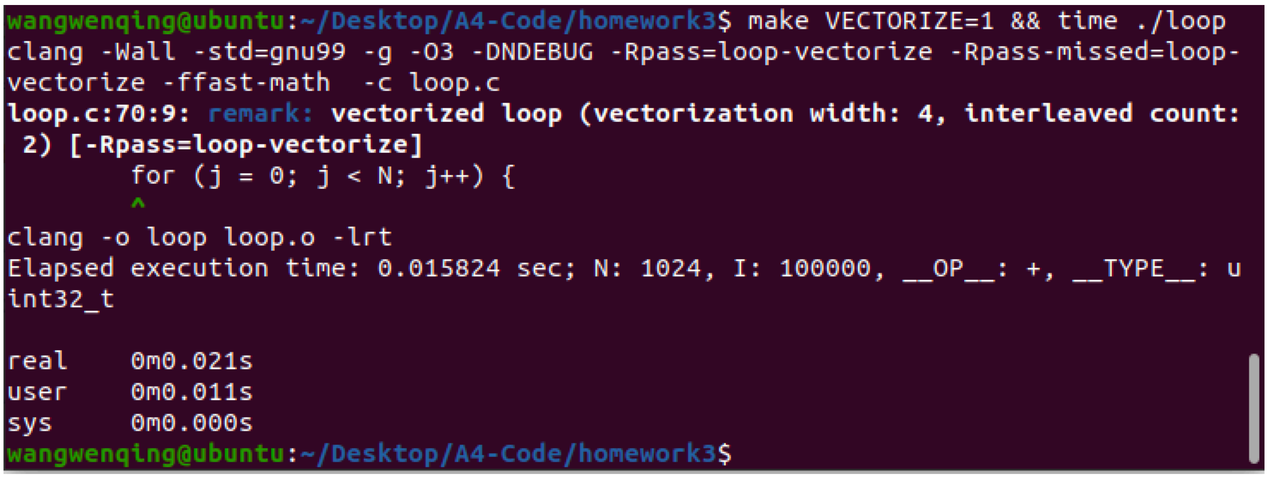
Write-up 6: What speedup does the vectorized code achieve over the unvectorized code? What additional speedup does using -mavx2 give? You may wish to run this experiment several times and take median elapsed times; you can report answers to the nearest 100% (e.g., 2×, 3×, etc). What can you infer about 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.



```
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ make
clang -Wall -std=gnu99 -g -O3 -DNDEBUG -fno-vectorize  -c loop.c
clang -o loop loop.o -lrt
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ ls
fasttime.h  loop  loop.c  loop.o  Makefile
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ time ./loop
Elapsed execution time: 0.054123 sec; N: 1024, I: 100000, __OP__: +, __TYPE__: u
int32_t

real	0m0.056s
user	0m0.043s
sys	0m0.004s
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ 
```

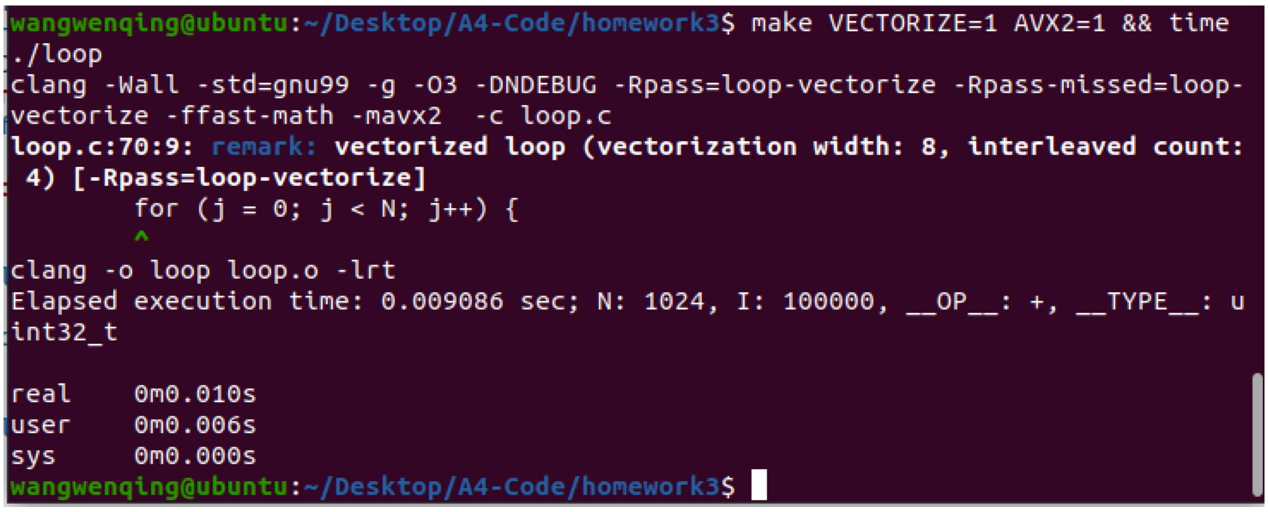
make VECTORIZE=1 && time ./loop



```
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ make VECTORIZE=1 && time ./loop
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
Elapsed execution time: 0.015824 sec; N: 1024, I: 100000, __OP__: +, __TYPE__: u
int32_t

real	0m0.021s
user	0m0.011s
sys	0m0.000s
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$
```

make VECTORIZE=1 AVX2=1 && time ./loop



```
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ make VECTORIZE=1 AVX2=1 && time
./loop
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
Elapsed execution time: 0.009086 sec; N: 1024, I: 100000, __OP__: +, __TYPE__: u
int32_t

real    0m0.010s
user    0m0.006s
sys     0m0.000s
wangwenqing@ubuntu:~/Desktop/A4-Code/homework3$ 
```

vectorized 的速度比 unvectorized快了4倍左右，因此其vector register 可能是128 bit

AVX2的速度 比 vectorized 快了2倍，因此其vector register 可能是256 bit