

# Homework 3 - Vectorization

Due: Feb 15 @ 4:55pm on [Gradescope](#)

Deliverables: Code and writeup

This homework is a hands-on introduction to Intel Vector Extensions. You will learn how to vectorize your code, figure out when vectorization has succeeded and debug when vectorization seems to have worked but you aren't seeing speedup.

Your homework should be completed **individually** (not in groups).

Code and some parts of the handout are from [MIT 6.172](#) and [Brian Wheatman](#).

All instructions in this handout assume that we are running on **PACE ICE**. Instructions for logging in are in HW1.

## Setup

### Get the code from git

The starter code is available on Github and should work out of the box. To get started, we recommend you log in to PACE ICE and download the code:

```
$ git clone https://github.com/cse6230-spring24/hw3.git
```

### Homework submission instructions

There are two parts to the homework: a writeup and the code. The writeup should be in pdf form, and the code should be in zip form. **There will be two submission slots** in Gradescope called "Homework 3 - writeup" and "Homework 3 - code". You should submit the writeup and code separately in their respective slots.

To get the code from PACE ICE to your local machine to submit, first zip it up on PACE ICE:

```
$ zip -r hw3.zip hw3/
```

Then from the terminal (or terminal equivalent) your local machine (e.g., your personal laptop), scp the code over.

```
$ scp <your-username-here>@login-ice.pace.gatech.edu:<path-to-hw2.zip>  
<local-file-location>
```

For example, if I had `hw3.zip` in my home directory on PACE ICE and wanted to copy it into my current directory on my local machine, I would run:

```
$ scp hxu615@login-ice.pace.gatech.edu:~/hw3.zip .
```

## Part 1: Loops tutorial

Consider a loop that performs element-wise addition between two arrays A and B, storing the result in array C. This loop is data parallel because the operation during any iteration  $i_1$  is independent of the operation during any iteration  $i_2$  where  $i_1 \neq i_2$ . In short, the compiler should be allowed to schedule each iteration in any order, or pack multiple iterations into a single clock cycle. The first option is covered by thread-level parallelism (e.g., via OpenMP). The second case is covered by vectorization, also known as “single instruction, multiple data” or SIMD.

Vectorization is a delicate operation: very small changes to loop structure may cause gcc to give up and not vectorize at all, or to vectorize your code but not yield the expected speedup. Occasionally, unvectorized code may be faster than vectorized code. Before we can understand this fragility, we must get a handle on how to interpret what gcc is actually doing when it vectorizes code; in Part 2, you will see the actual performance impacts of vectorizing code.

### Example 1

We will start with the following simple loop:

```
#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];
    }
}
```

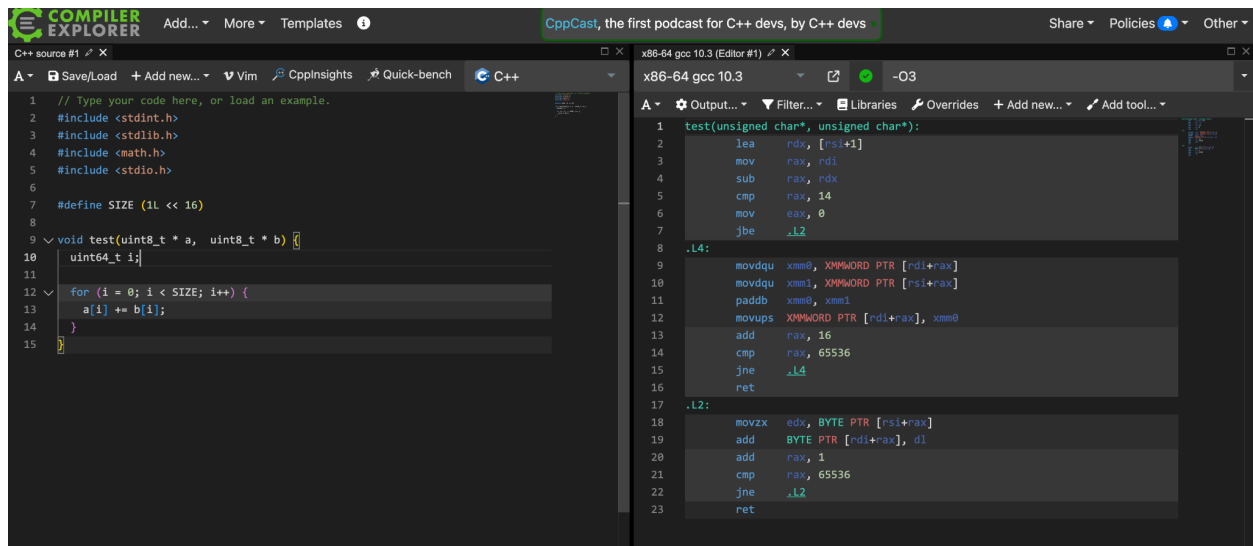
Although gcc automatically vectorizes at -O3, you should always look at the assembly to see exactly how it has been vectorized, since it is not guaranteed to be using the vector registers optimally. Here is a [guide](#) to reading x86 assembly, and here is an [x86 instruction set reference](#).

To view the assembly, we will use the [Godbolt compiler explorer](https://godbolt.com). Compiler Explorer is an interactive online compiler which shows the assembly output of compiled C/C++ (among many other languages). It also provides helpful highlighting to show which parts of the original code correspond to which part of the assembly.

Here is the assembly if we turn off vectorization using the `-fno-tree-vectorize` flag:  
<https://godbolt.org/z/cbh8TjYhc>

By default, this code is vectorized to some extent on `-O3`:  
<https://godbolt.org/z/sfch95T4f>

You should see something like this:



To learn more about each assembly instruction, hover over it with the mouse.

Looking at the assembly code, we can see that this code first checks if there is a partial memory location overlap between array `a` and `b`. If there is an overlap, then it does a simple non-vectorized code (`.L2`). If there is no overlap, then it can do a vectorized version (`.L4`). The above can, at best, be called partially vectorized.

The problem is that the compiler is constrained by what we tell it about the arrays. If we tell it more, then perhaps it can do more optimization. The most obvious thing is to inform the compiler that no overlap is possible. This is done in standard C by using the `restrict` qualifier for the pointers.

Modify the C code in Godbolt to match the following:

```
void test(uint8_t *__restrict a, uint8_t *__restrict b) {  
    uint64_t i;  
  
    for (i = 0; i < SIZE; i++) {  
        a[i] += b[i];  
    }  
}
```

Now you should see the following assembly:

```
test(unsigned char*, unsigned char*):  
    xor     eax, eax  
.L2:  
    movdqu xmm0, XMMWORD PTR [rdi+rax]  
    movdqu xmm1, XMMWORD PTR [rsi+rax]  
    paddb  xmm0, xmm1  
    movups XMMWORD PTR [rdi+rax], xmm0  
    add    rax, 16  
    cmp    rax, 65536  
    jne    .L2  
    ret
```

The generated code is better, but it is assuming the data are NOT 16 bytes aligned (movdqu is unaligned move). It also means that the loop above can not assume that both arrays are aligned.

If gcc were smart, it could test for the cases where the arrays are either both aligned, or both unaligned, and have a fast inner loop. However, it does not do that currently. So in order to get the performance we are looking for, we need to tell clang that the arrays are aligned. There are a couple of ways to do that. The first is to construct a (non-portable) aligned type, and use that in the function interface. The second is to add an intrinsic or two within the function itself. The second option is easier to implement on older code bases, as other functions calling the one to be vectorized do not have to be modified. The intrinsic gcc has for this is called `__builtin_assume_aligned`:

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

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

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

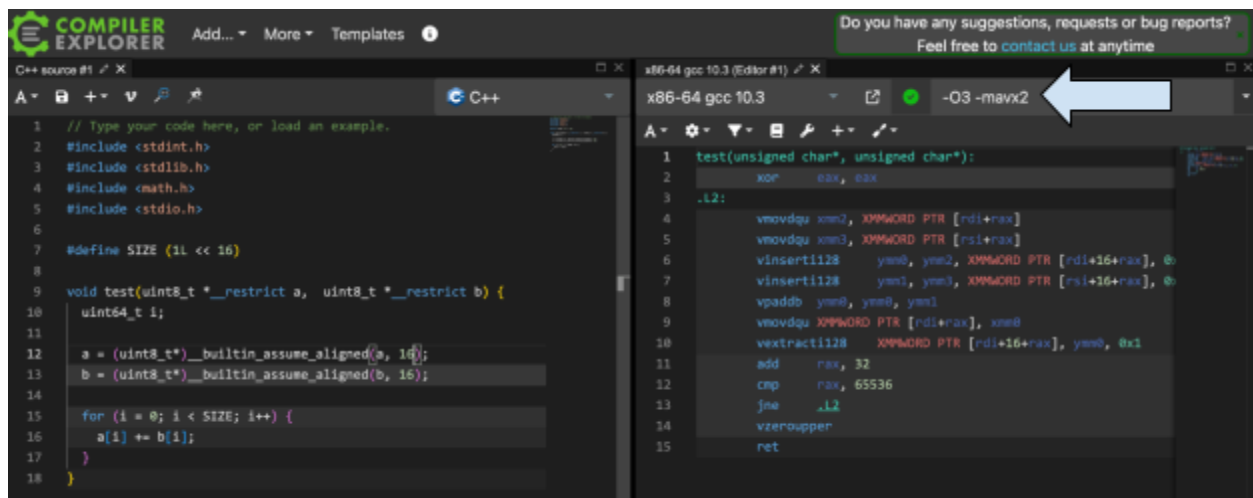
After you add the instruction `__builtin_assume_aligned`, you should see something similar to the following output:

```
test(unsigned char*, unsigned char*):
    xor     eax, eax
.L2:
    movdqa  xmm0, XMMWORD PTR [rdi+rax]
    paddb   xmm0, XMMWORD PTR [rsi+rax]
    movaps  XMMWORD PTR [rdi+rax], xmm0
    add     rax, 16
    cmp     rax, 65536
    jne     .L2
    ret
```

Now finally, we get the nice tight vectorized code (`movdqa` is aligned move) we were looking for, because gcc has used packed SSE instructions to add 16 bytes at a time. It also manages to load and store two at a time, which it did not do last time.

(As an alternative, in C++, there is a portable [assume\\_aligned](#) function in STD).

Next, we try to turn on AVX2 instructions by adding the flag `-mavx2` to the compiler flags in Godbolt:



```
COMPILER EXPLORER
Add... More Templates
C++ source #1
1 // Type your code here, or load an example.
2 #include <stdint.h>
3 #include <stdlib.h>
4 #include <math.h>
5 #include <stdio.h>
6
7 #define SIZE (1L << 16)
8
9 void test(uint8_t *__restrict a, uint8_t *__restrict b) {
10     uint64_t i;
11
12     a = (uint8_t*)__builtin_assume_aligned(a, 16);
13     b = (uint8_t*)__builtin_assume_aligned(b, 16);
14
15     for (i = 0; i < SIZE; i++) {
16         a[i] += b[i];
17     }
18 }
x86-64 gcc 10.3 (Editor #1)
x86-64 gcc 10.3 -O3 -mavx2
1 test(unsigned char*, unsigned char*):
2     xor     eax, eax
3     .L2:
4     vmovdqu xmm2, XMMWORD PTR [rdi+rax]
5     vmovdqu xmm3, XMMWORD PTR [rsi+rax]
6     vinserti128 ymm0, ymm2, XMMWORD PTR [rdi+16+rax], 0
7     vinserti128 ymm1, ymm3, XMMWORD PTR [rsi+16+rax], 0
8     vpaddb ymm0, ymm0, ymm1
9     vmovdqu XMMWORD PTR [rdi+rax], xmm0
10    vextracti128 XMMWORD PTR [rdi+16+rax], ymm0, 0x1
11    add     rax, 32
12    cmp     rax, 65536
13    jne    .L2
14    vzeroupper
15    ret
```

**Write-up 1:** This code is still not aligned when using AVX2 registers. Fix the code to make sure it uses aligned moves. How did you change the code (Hint: change the alignment constant in `__builtin_assume_aligned`)?

**Note:** There may not be much performance difference between unaligned and aligned instructions on some architectures. For example, here are the latency and throughput on recent Intel architectures for [unaligned](#) and [aligned](#) instructions.

Unaligned:

---

`__m256i _mm256_lddqu_si256 (__m256i const * mem_addr)` vlddqu

`__m256i _mm256_loadu_si256 (__m256i const * mem_addr)` vmovdqu

**Synopsis**

```
__m256i _mm256_loadu_si256 (__m256i const * mem_addr)
#include <immintrin.h>
Instruction: vmovdqu ymm, m256
CPUID Flags: AVX
```

**Description**

Load 256-bits of integer data from memory into `dst.mem_addr` does not need to be aligned on any particular boundary.

**Operation**

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

**Latency and Throughput**

Architecture	Latency	Throughput (CPI)
Alderlake	7	0.333333333
Icelake Intel Core	7	0.5
Icelake Xeon	7	0.56
Sapphire Rapids	7	0.333333333
Skylake	7	0.5

Aligned:

---

`__m256i _mm256_load_si256 (__m256i const * mem_addr)` vmovdqa

**Synopsis**

```
__m256i _mm256_load_si256 (__m256i const * mem_addr)
#include <immintrin.h>
Instruction: vmovdqa ymm, m256
CPUID Flags: AVX
```

**Description**

Load 256-bits of integer data from memory into `dst.mem_addr` must be aligned on a 32-byte boundary or a general-protection exception may be generated.

**Operation**

```
dst[255:0] := MEM[mem_addr+255:mem_addr]
dst[MAX:256] := 0
```

**Latency and Throughput**

Architecture	Latency	Throughput (CPI)
Alderlake	7	0.333333333
Icelake Intel Core	7	0.5
Icelake Xeon	7	0.56
Sapphire Rapids	7	0.333333333
Skylake	7	0.5

---

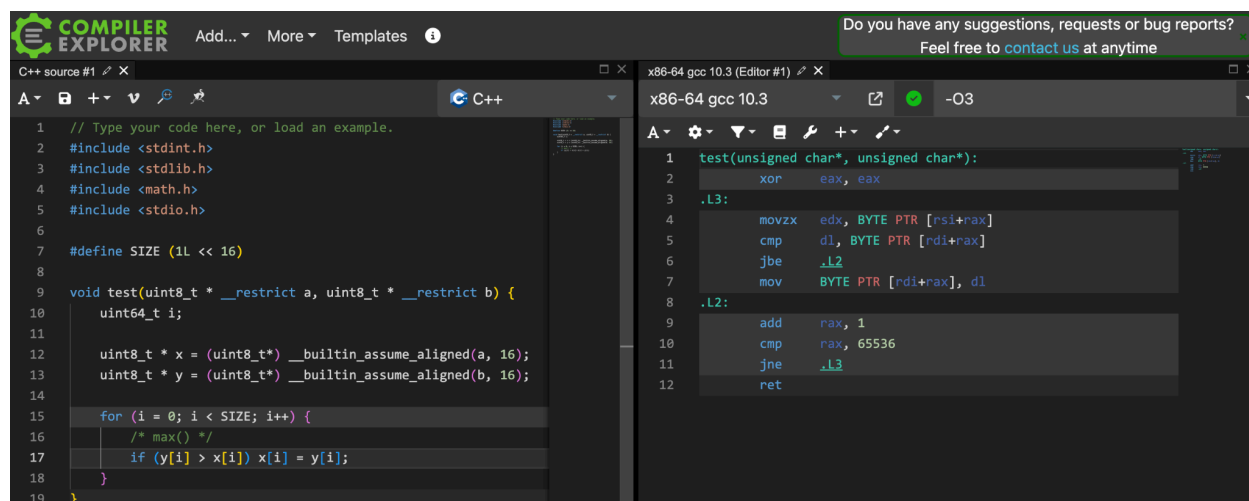
`__m256i _mm256_stream_load_si256 (void const* mem_addr)` vmovntdqa



## Example 2

Now that we understand what we need to tell the compiler, how much more complex can the loop be before auto-vectorization fails?

Let's look at another example (Godbolt here: <https://godbolt.org/z/bfKKz5snW>)



```
1 // Type your code here, or load an example.
2 #include <stdint.h>
3 #include <stdlib.h>
4 #include <math.h>
5 #include <stdio.h>
6
7 #define SIZE (1L << 16)
8
9 void test(uint8_t * __restrict a, uint8_t * __restrict b) {
10     uint64_t i;
11
12     uint8_t * x = (uint8_t*) __builtin_assume_aligned(a, 16);
13     uint8_t * y = (uint8_t*) __builtin_assume_aligned(b, 16);
14
15     for (i = 0; i < SIZE; i++) {
16         /* max() */
17         if (y[i] > x[i]) x[i] = y[i];
18     }
19 }
```

```
1 test(unsigned char*, unsigned char*):
2     xor     eax, eax
3 .L3:
4     movzx  edx, BYTE PTR [rsi+rax]
5     cmp    dl, BYTE PTR [rdi+rax]
6     jbe   .L2
7     mov   BYTE PTR [rdi+rax], dl
8 .L2:
9     add   rax, 1
10    cmp   rax, 65536
11    jne   .L3
12    ret
```

Note that the assembly does not vectorize nicely.

**Write-up 2:** Modify the code so that it vectorizes nicely (Hint: use a ternary operator so that each iteration will write to the output).

### Example 3

Let's look at another example loop (Godbolt here: <https://godbolt.org/z/53r5n7Y7f>):

```
void test(uint8_t * __restrict a, uint8_t * __restrict b) {  
    uint64_t i;  
  
    for (i = 0; i < SIZE; i++) {  
        a[i] = b[i+1];  
    }  
}
```

And the assembly:

```
test(unsigned char*, unsigned char*):  
    add     rsi, 1  
    mov     edx, 65536  
    jmp     memcpy
```

**Write-up 3:** 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.

## Example 4

Take a look at the following loop (Godbolt here: <https://godbolt.org/z/ofx6q6c31>):

```
double test(double * __restrict a) {
    size_t i;

    double *x = (double*) __builtin_assume_aligned(a, 16);

    double y = 0;

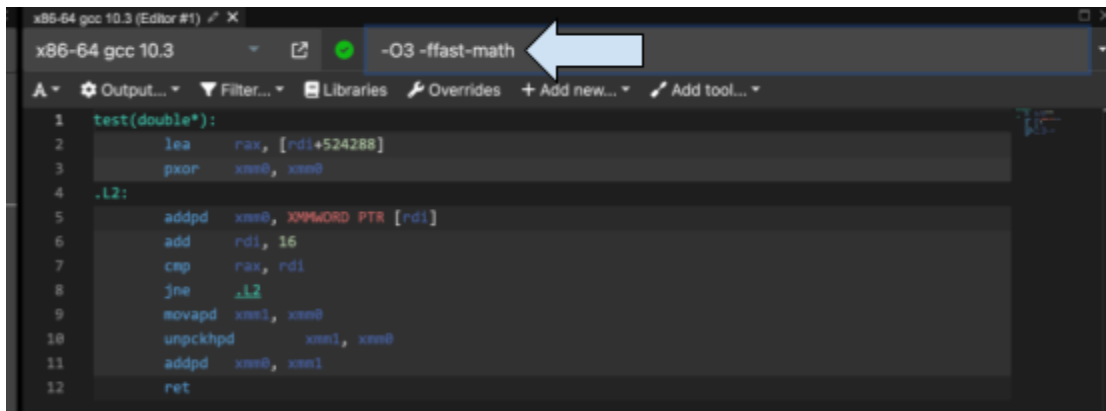
    for (i = 0; i < SIZE; i++) {
        y += x[i];
    }
    return y;
}
```

You should see the non-vectorized code with the `addsd` instruction.

```
test(double*):
    lea    rax, [rdi+524288]
    pxor   xmm0, xmm0
.L2:
    movsd  xmm1, QWORD PTR [rdi]
    add    rdi, 16
    addsd  xmm1, xmm0
    movsd  xmm0, QWORD PTR [rdi-8]
    addsd  xmm0, xmm1
    cmp    rax, rdi
    jne    .L2
    ret
```

Notice that this does not actually vectorize as the `xmm` registers are operating on 8 byte chunks. The problem here is that `gcc` is not allowed to re-order the operations we give it. Even though the addition operation is associative with real numbers, they are not with floating point numbers (e.g., due to roundoff error).

Furthermore, we need to tell `clang` that reordering operations is okay with us. To do this, we need to add another compile-time flag, `-ffast-math`. Add the compilation flag `-ffast-math` to Godbolt and recompile.



```
x86-64 gcc 10.3 (Editor #1)
x86-64 gcc 10.3 -O3 -ffast-math
Output... Filter... Libraries Overrides + Add new... Add tool...
1 test(double*):
2   lea   rax, [rdi+524288]
3   pxor  xmm0, xmm0
4   .L2:
5   addpd xmm0, XMMWORD PTR [rdi]
6   add   rdi, 16
7   cmp   rax, rdi
8   jne   .L2
9   movapd xmm1, xmm0
10  unpcckhpd xmm1, xmm0
11  addpd  xmm0, xmm1
12  ret
```

Check the assembly and verify that it does vectorize properly. Here is the output from running with and without `-ffast-math` (example4.c is in hw3/ from git):

```
[hxu615@login-ice-3 hw3]$ gcc -O3 example4.c -o example4
[hxu615@login-ice-3 hw3]$ ./example4
The decimal floating point sum result is: 11.667578
The raw floating point sum result is: 0x1.755cccec10aa5p+3

[hxu615@login-ice-3 hw3]$ gcc -O3 --ffast-math example4.c -o example4
[hxu615@login-ice-3 hw3]$ ./example4
The decimal floating point sum result is: 11.667578
The raw floating point sum result is: 0x1.755cccec10a96p+3
```

**Write-up 4:** What do you notice about the two outputs? Explain why they are different.

## Example 5

A simplifying feature in our loop is that its stride (or step) equals 1. Stride corresponds to how big our steps through the array are; e.g., `j++`, `j+=2`, etc.

Let's look at an example with a different stride (Godbolt here: <https://godbolt.org/z/K14cM1G33>):

```
void test(uint32_t * __restrict a, uint32_t * __restrict b, uint32_t * __restrict c) {
    uint64_t i;

    uint32_t * x = (uint32_t*) __builtin_assume_aligned(a, 32);
    uint32_t * y = (uint32_t*) __builtin_assume_aligned(b, 32);
    uint32_t * z = (uint32_t*) __builtin_assume_aligned(c, 32);

    for (i = 0; i < SIZE; i+=2) {
        z[i] = x[i] + y[i];
    }
}
```

Look at the assembly in Godbolt.

**Write-up 5:** Does gcc vectorize the code with this strided loop? Why might it choose not to vectorize the code?

## Example 6

A very common operation is to combine elements in an array (somehow) into a single value. For instance, one might wish to sum up the elements in an array. For example (Godbolt here: <https://godbolt.org/z/K7d6exnKy>):

```
uint64_t test(uint32_t * __restrict a) {  
    uint64_t i;  
    uint32_t total = 0;  
  
    for (i = 0; i < SIZE; i++) {  
        total += a[i];  
    }  
    return total;  
}
```

The compiler can implement reduction via a technique called [strip mining](#).

**Write-up 6:** This code vectorizes, but how does it vectorize? Look at the assembly in Godbolt, and explain what the compiler is doing.

As discussed in lecture, this reduction will only vectorize if the combination operation (+) is associative.

## Part 2: Writing vectorized code

In this part you will tackle a new problem, write some code for it, and then analyze it. The problem can be found at <https://godbolt.org/z/T7orGs84a>

The goal of the problem is to count occurrences of bytes in an array.

Here is the starting code:

```
long CountBytes(uint8_t *data, long n, uint8_t target) {
    long count = 0;
    for (long i = 0; i < n; i++) {
        if (data[i] == target) {
            count += 1;
        }
    }
    return count;
}
```

To build it with the test driver, go to the hw3/ repository and run

```
$ g++ -O3 -mavx2 count_bytes.cpp -o count_bytes
$ ./count_bytes
```

Make sure you have allocated an interactive node with at least one core before you run the code.

You should see output like the following:

```
$ Time per trial: 0.359268 seconds, got 4196416 as the count.
```

How to view assembly in the terminal

To inspect the assembly code for count\_bytes.cpp, run the following

```
$ g++ -S -O3 -g -o count_bytes.s count_bytes.cpp
```

Now, let's inspect the assembly code in count\_bytes.s, which should give you the assembly without line annotations.

To get the assembly with line annotations:

```
$ as -alhnd count_bytes.s > count_bytes.lst
```

Now count\_bytes.lst should have the assembly with corresponding lines from the original C++.

## Optimizing the code

**Writeup 7:** Please achieve at least 2x speedup over the reference code **with one thread using intrinsics** and include your code in your homework submission.

Since the vector registers are 256 bits and we are processing bytes (8-bit elements), each iteration of the inner loop should process 32 elements.

Report the runtime before and after implementing the vectorized version.

You must explain your solution in English as well. Submissions without a full explanation will not receive points.

Explanations should include:

- how does it compute the answer?
- how many iterations of the base loop from the starting code does it compute on each iteration?

For documentation about intrinsics, check the [Intel Intrinsics Guide](#).

That's the end of this homework! Submit your writeup and code as described in the "Homework submission instructions" to Gradescope.



+

