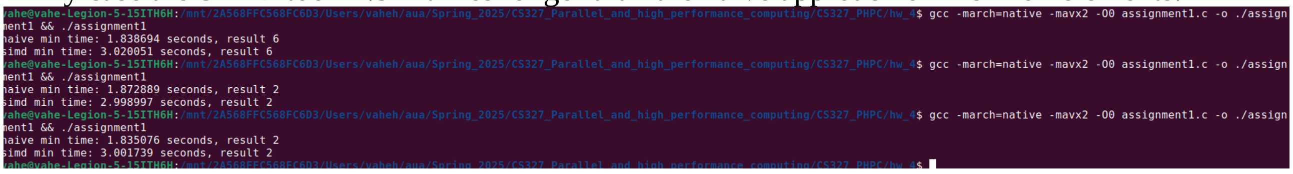
Assignment 1: Finding min

I create a new vector vmin and assign to it the first 8 elements of the array
Then iterate over the rest of the array, get next 8 elements, calculate the min of the vmin and the current 8 elements, and store the result in vmin.

Do this for the tailing 8 elements after the loop. Since we're looking for the min element here, it doesn't matter if I process a few of the last elements twice.

Then at the very end iterate over the 8 elements of vmin and return the minimal using naive logic.

In my case the SIMD took 1.5-2 times longer than the naive approach on 1 billion elements.



```
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ gcc -march=native -mavx2 -O0 assignment1.c -o ./assign
ment1 && ./assignment1
naive min time: 1.838694 seconds, result 6
simd min time: 3.020051 seconds, result 6
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ gcc -march=native -mavx2 -O0 assignment1.c -o ./assign
ment1 && ./assignment1
naive min time: 1.872889 seconds, result 2
simd min time: 2.998997 seconds, result 2
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ gcc -march=native -mavx2 -O0 assignment1.c -o ./assign
ment1 && ./assignment1
naive min time: 1.835076 seconds, result 2
simd min time: 3.001739 seconds, result 2
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$
```

Assignment 2: Convolution

The way I used SIMD here is the following

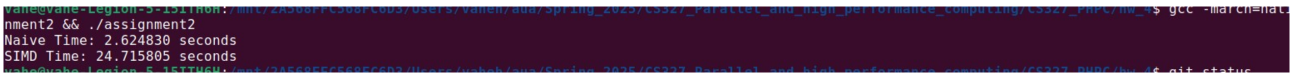
Calculate a[i] * weight1, and store it in an array
Calculate a[i] * weight2, store it
Calculate a[i] * weight3, store it

Note that for weight 1, I calculate from 0 to SIZE - 2
for weight 2: 1 to SIZE - 1
and weight 3: 2 to SIZE
because I leave out the first and last element in the calculation of the convolution. The resulting array has size SIZE - 2

I intentionally calculate this way so memory access will be almost sequential (always iterate to the +8th index).

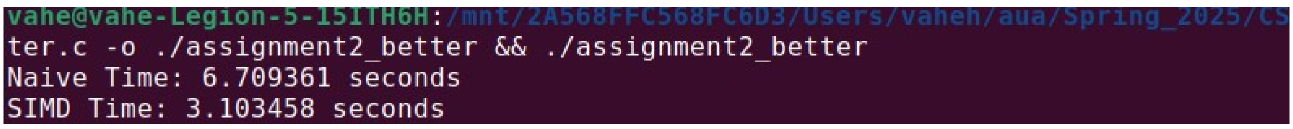
Then I just add the result of weight 1 to result of weight 2
Then to that I add the previous sum to weight 3 result.

But, again SIMD version works much much slower for me. I'm not sure if this is because of my pc or my code is just bad



```
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ gcc -march=nati
nment2 && ./assignment2
Naive Time: 2.624830 seconds
SIMD Time: 24.715805 seconds
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ git status
```

Ok turned out my code was bad in this case. I found a better way to do it with much less loading and saving



```
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS
ter.c -o ./assignment2_better && ./assignment2_better
Naive Time: 6.709361 seconds
SIMD Time: 3.103458 seconds
```

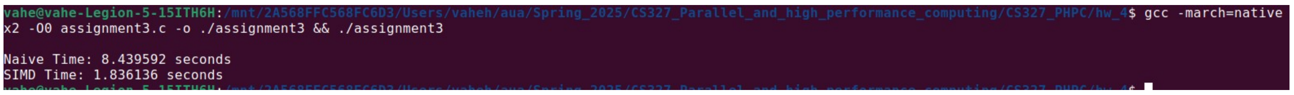
Assignment 3: Nothing too special in this case. I intentionally used the same logic for both naive and SIMD approaches - for every i, start from 0 up to i every time.

For SIMD, I keep a __m256i register for 0s only, and I don't modify it.

For every iteration I copy these 0s to the result of the current iteration (cr), load the next 8 integers from a, add the two (cr + ca) and store it in cr itself.

In the end of each iteration I add all the rest of entries from a, that we didn't cover with SIMD, and also add the resulting 8 integers from cr.

The results are quite good for 100000 elements



```
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ gcc -march=native
x2 -O0 assignment3.c -o ./assignment3 && ./assignment3

Naive Time: 8.439592 seconds
SIMD Time: 1.836136 seconds
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$
```

Assignment 4: Matrix multiplication

At first my idea was to do as few load operations as possible. If in common way we do the calculation going over all columns of each row then going to the next row, we get 2 load operations per 8 ints: 1 for the 8 ints of the current row of matrix, and 1 for the 8 ints of the vector, because on the next iteration we need the next 8 ints.

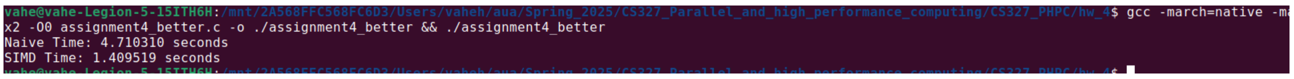
My strategy was instead of doing that, to go row by row, taking the 8 ints of each row, multiplying that with the 8 elements of the vector, adding that to the result, then going to the next row. This would mean that I wouldn't need to load the 8 ints of the vector on each iteration.

But this surprisingly made things much slower
1. Because after multiplying each 8 ints, I had to iterate over them, add them up.
2. I assume because memory access is much slower this way. I was hoping 1 less load operation would mitigate this, but it did not.

If you're interested, you can check the assignment4_bad.c file.

Then I just tried everything normally. Just load 8 ints from current row, multiply by 8 ints from vector, then take next 8 ints from current row and multiply that with next 8 ints of vector, and so on.

This resulted in code much faster than naive approach. Please check assignment4_better.c



```
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$ gcc -march=native -ma
x2 -O0 assignment4_better.c -o ./assignment4_better && ./assignment4_better
Naive Time: 4.710310 seconds
SIMD Time: 1.409519 seconds
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PHPC/hw_4$
```

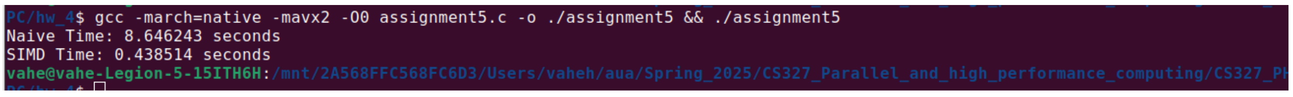
Assignment5: Uppercase

Nothing too special here. The code does what assignment required.

I first create 3 vectors that I will use throughout the algorithm - vectors of 96s, 123s and 32s.
Then in the loop I load the next 32 chars from the array, then compare that to the 96s vector with cmpgt, and do the same with 123s vector, just the other way around, so I get less than.

The result of cmpgt is 1s for the whole 1 byte where the condition matched. I AND these two results, and since I only need the 5th bit of each char, I then AND the result of that with the vector of 32s. Then the result is XOR-ed with original chars, which gives us the desired result.

The results of this one was much much faster, about 20 times. This is the time for 500000000 length string



```
PC/hw_4$ gcc -march=native -mavx2 -O0 assignment5.c -o ./assignment5 && ./assignment5
Naive Time: 8.646243 seconds
SIMD Time: 0.438514 seconds
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PH
PC/hw_4$
```

Assignment6: Grayscale

I will start by mentioning that the reading the bmp file and writing it to the disk is done almost entirely by prompting ChatGPT. I asked for some modifications, like storing the rgb lanes into separate 3 arrays, so I can process them easily, but most of it is written by ChatGPT

How SIMD is done. First, my plan is to avoid having to deal with float values and divisions inside my SIMD code. Instead of keeping the 0.299, 0.587, 0.114 values as floats in some way, I'll make them 16 bit ints (why 16, I'll explain in a bit). I will use bit shift to convert everything back.

Now, we're going to lose some precision here, but we were going to lose this precision either way, because our formula for grayscale calculates a float number, then converts it to int.

Let's chose some x, in such a way that we find the maximum possible value for this

Max(0.299, 0.587, 0.114) * 2^x * (maximum value for any lane) < 2^16 - 1
=
0.587 * 2^x * 255 < 2^16 - 1
2^x < 437.8..
so x is 8, 2^x = 256

We find the maximum just so we lose minimal precision.

So I use 3 256 bit registers, all of them filled with the floored values of
floor(0.299 * 256) = 76
floor(0.587 * 256) = 150
floor(0.114 * 256) = 29

And I keep each int in 16 bits already.

So in the loop for actually processing the pixels. I read from red, green, blue each 32 8 bit ints, then I know I'm going to need to multiply those values by the numbers we calculated earlier. And I'm sure that the multiplication is not going to overflow if I use 16 bits.

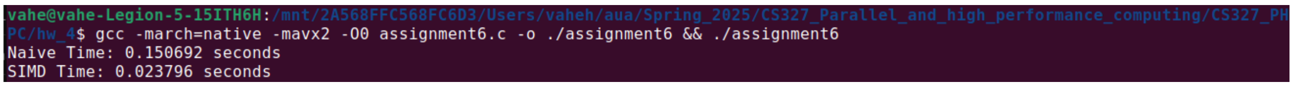
So I take my 8 bit ints, unpack them to two separate 16 bit int registers. Then I multiply these 16 bit ints with my 16 bit constants, then add them up.

Then I shift to the right both of these registers by 8 bits to divide by 256 because remember I multiplied the constants by 256.

And since I need to write back to my resulting array 8 bit ints, I pack the two 16 bit int registers into 8 bit int register, then write it to the result.

The results of this one are actually quite nice. I got ~7-8 times faster results with SIMD.

Here's the timing for 5000 x 5000 pixel bmp file.



```
vahe@vahe-Legion-5-15ITH6H:/mnt/2A568FFC568FC6D3/Users/vaheh/aua/Spring_2025/CS327_Parallel_and_high_performance_computing/CS327_PH
PC/hw_4$ gcc -march=native -mavx2 -O0 assignment6.c -o ./assignment6 && ./assignment6
Naive Time: 0.150692 seconds
SIMD Time: 0.023796 seconds
```

This was a nice problem, thank you!