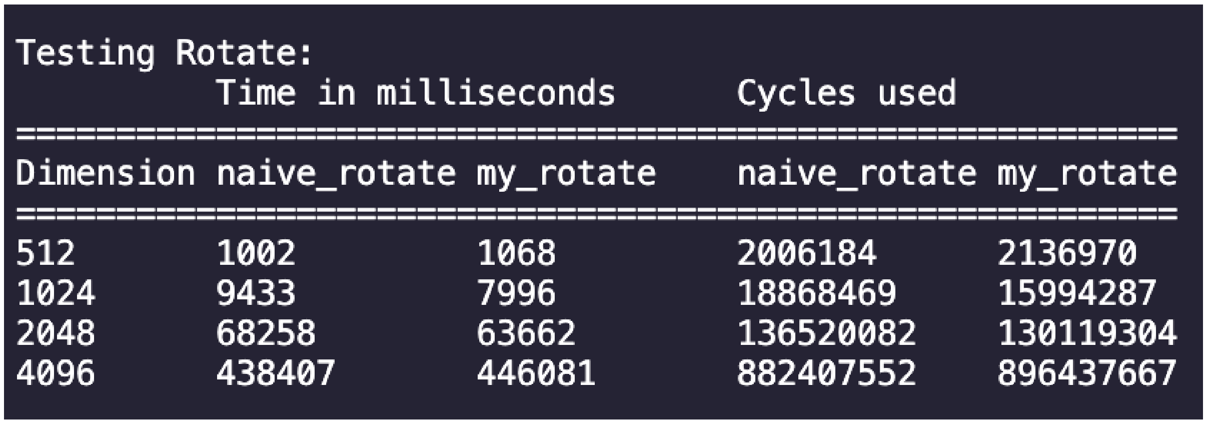
Project 6 Report

When trying to optimize the two functions I tried multiple different optimization techniques. Starting with the rotate function, at first I tried to just reduce the number of operations needed to be done by doing the calculation for RIDX directly in the index of dst and src, I also tried saving the RIDX to a variable first then putting that variable in the index of dst and src. Both of these resulted in the program not being able to run and many errors coming up. I next tried using pointers to access the elements dst and src instead of using RIDX, my thought was that using pointers would be more efficient since this allows the compiler to generate more efficient machine code and avoid the overhead of calling a function. This resulted in better scores for dimensions 1024 and 2048, but worse scores for dimensions 512 and 4096. The results of this attempt are in the table below.



```
Testing Rotate:
          Time in milliseconds      Cycles used
==========================================================
Dimension naive_rotate my_rotate    naive_rotate my_rotate
==========================================================
512       1002         1068         2006184      2136970
1024      9433         7996         18868469     15994287
2048      68258        63662        136520082    130119304
4096      438407       446081       882407552    896437667
```

After this I read the specs again and looked through the code to get a better understanding of what was going on. For my final attempt I added a nested for loop in the existing nested for loop that did all the calculations and changes in intervals of 16. I had trouble getting the order and getting the calculations right, but in the end I didn't call RIDX and did all the calculations in the loops. The code is a nested loop that performs an operation on a two-dimensional array, src, and stores the result in another two-dimensional array, dst. The outermost loop iterates over src in blocks of 16x16 elements, and the next two loops iterate over each block, processing elements one at a time. At each iteration of the innermost loop, the code calculates the index into dst using the formula d - l*dim, where l is the current iteration index of the innermost loop, and then sets the element at this index equal to the element at index dim2 + l in src. This resulted in a 52% cycle score improvement amongst the average of all dimensions. The resulting scores are in the table below.

Dimension	naive_rotate()	my_rotate()	Improvement
512	2173183	1765608	1.2308 (~23%)
1024	13875366	8747583	1.5862 (~59%)
2048	113433730	83748901	1.3545 (~35%)
4096	940947569	488886372	1.9247 (~92%)

For the smooth function I was able to get a better score by reversing the order of the for loops, but to take it another step I did the calculations for avg() and RIDX in the loop itself so I didn't have to waste time calling avg() and searching for RIDX. I reverse the loops since the original way was inefficient from a grid point of view, it traveled too long to get to the next column, which we learned in class (row major access pattern). Sequential accesses instead of striding through memory, improved spatial locality. At the end I was able to get an improvement of 38.75% amongst the average of all dimensions. The resulting scores are in the table below.

Dimension	naive_rotate()	my_rotate()	Improvement
256	17226953	15661572	1.099 (~10%)
512	69246967	57687271	1.200 (20%)
1024	316709397	214401096	1.477 (~48%)
2048	1685254179	948732438	1.776 (~78%)

In the end I chose to go with these optimizations since they were the best I could come up with and provided a good amount of improvement in terms of cycles used. Opposed to the other optimizations I tried, these appeared to work and used as little function calls as necessary, while also doing the operations on only the necessary pixels in the 16x16 block for my_rotate() and changing the order for my_smooth(). I also used the row major access pattern optimization we learned in class to get better access for my_smooth().