Item 1

```
cs21FWX1@TL1PC08:~$ hostname
TL1PC08
cs21FWX1@TL1PC08:~$ lscpu | grep cache
L1d cache:                            256 KiB (8 instances)
L1i cache:                            256 KiB (8 instances)
L2 cache:                             4 MiB (8 instances)
L3 cache:                             16 MiB (1 instance)
cs21FWX1@TL1PC08:~$
```

Item 2

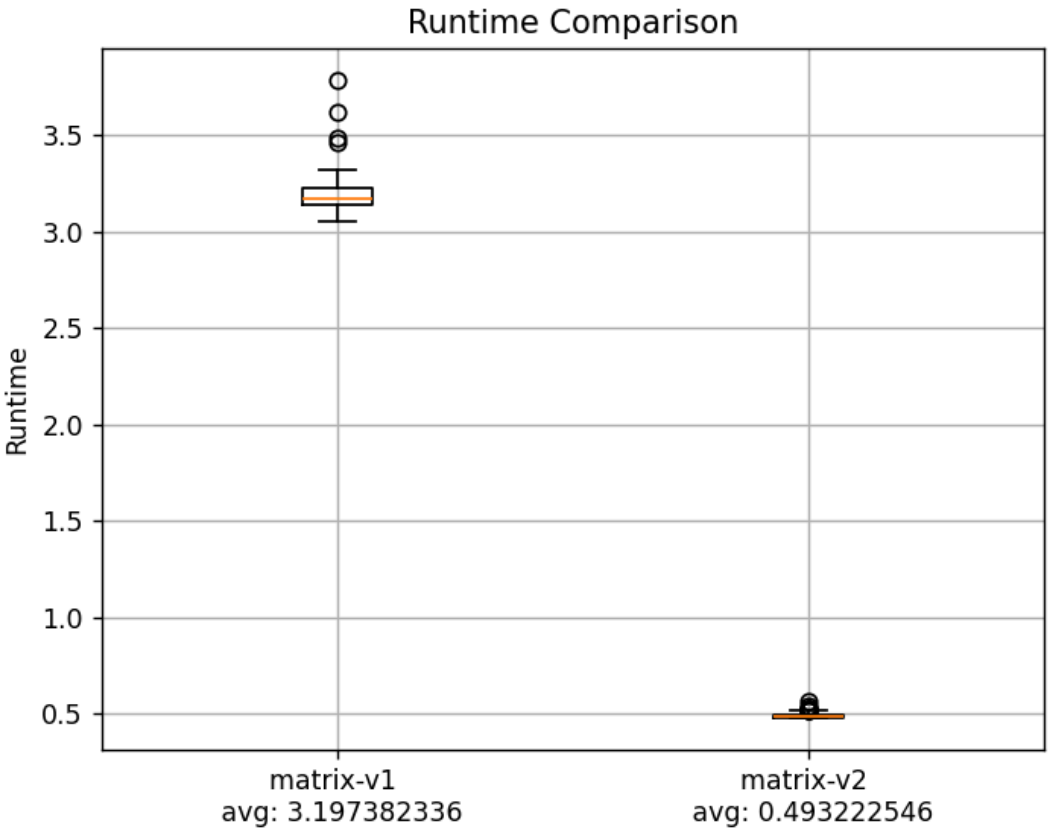
```
40      for (int i = 0; i < N; i++) {
41          for (int k = 0; k < N; k++) {
42              for (int j = 0; j < N; j++) {
43                  c[i][j] += a[i][k] * b[k][j];
44              }
45          }
46      }
```

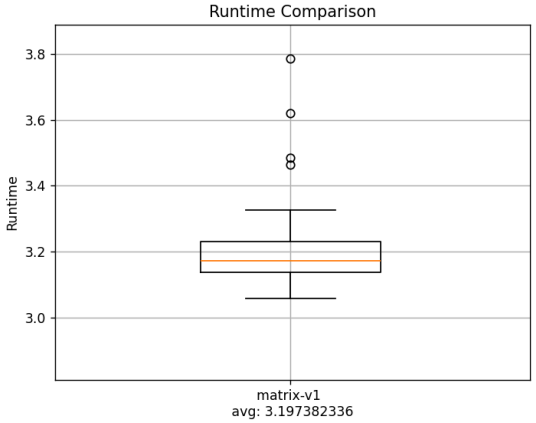
Looking at `a[i][k]` and `c[i][j]`, they just swapped locality.

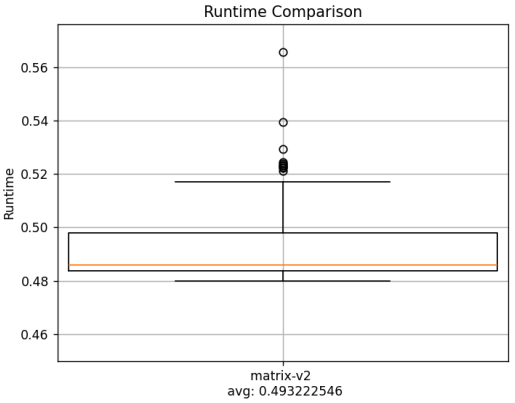
In the old version where `j` is nested right underneath `i` in the `for` loop, we would just remain at the same `c[i][j]` as we iterate through all the possible `k`s. In essence, as we walk through all `k` columns of `row i` of `matrix a`, we only write to `c[i][j]`. This time around with the swapped code, `k` is nested right underneath `i`. So as we walk through all `j` columns in row `i` of `matrix c`, we are just staying at `a[i][k]`.

The real improvement in locality is the order in how we iterate `b[k][j]`. In the old variation where the nested order is `i j k`, for each iteration, we could be switching the row of `matrix b` that we will need to read. Compare this to the swapped version where the nested order is `i k j`. We are now simply going through the `j columns` of the `row k`. This is faster because accessing the same row in succession is much better in terms of spatial locality compared to accessing different rows in succession.

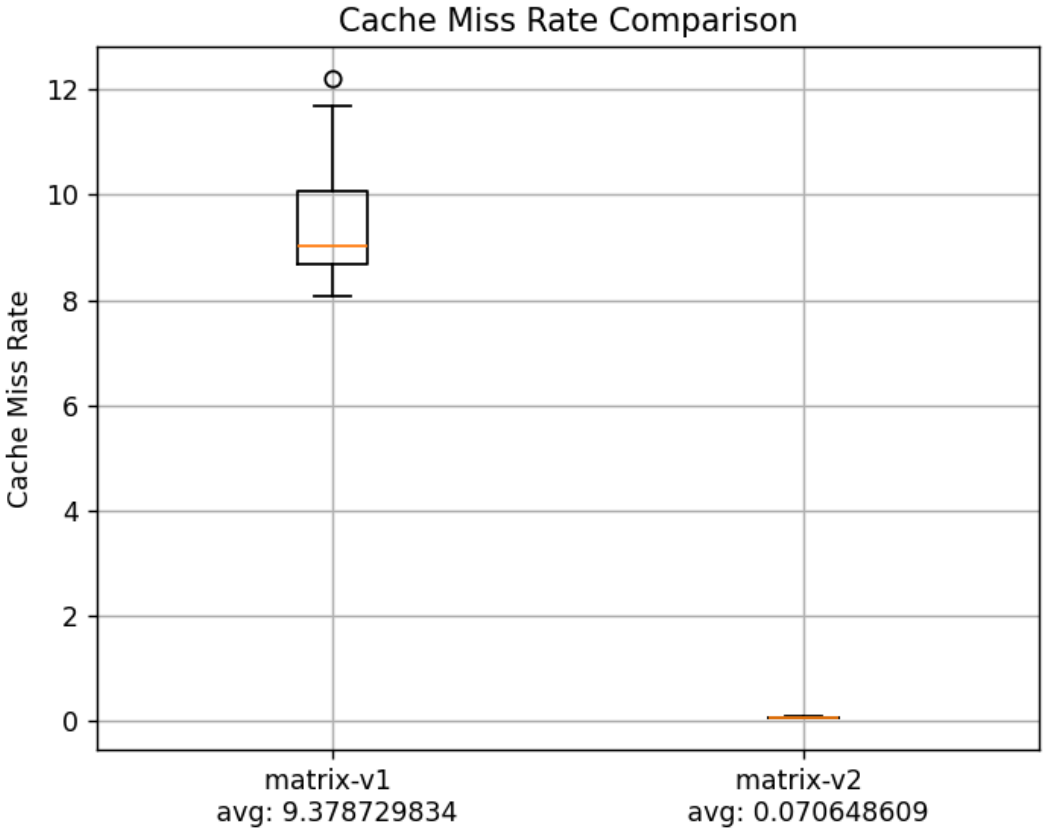
Item 3a

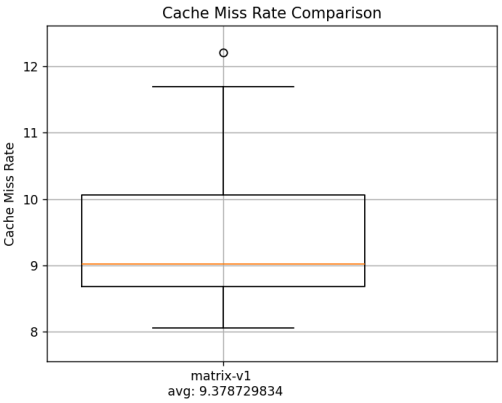

Runtime Comparison
Runtime
3.5
3.0
2.5
2.0
1.5
1.0
0.5
matrix-v1
avg: 3.197382336
matrix-v2
avg: 0.493222546

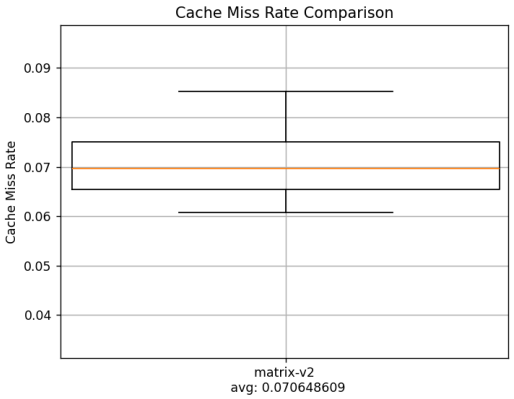

Runtime Comparison
Runtime
3.8
3.6
3.4
3.2
3.0
matrix-v1
avg: 3.197382336


Runtime Comparison
Runtime
0.56
0.54
0.52
0.50
0.48
0.46
matrix-v2
avg: 0.493222546

Item 3b


Cache Miss Rate Comparison
Cache Miss Rate
12
10
8
6
4
2
0
matrix-v1
avg: 9.378729834
matrix-v2
avg: 0.070648609


Cache Miss Rate Comparison
Cache Miss Rate
12
11
10
9
8
matrix-v1
avg: 9.378729834


Cache Miss Rate Comparison
Cache Miss Rate
0.09
0.08
0.07
0.06
0.05
0.04
matrix-v2
avg: 0.070648609

Item 3c

```
Shiro@LAPTOP-U2KIJPRU MINGW64 ~/cs21lab08 (master)
$ python item3c.py
Average time elapsed for matrix-v1 (in seconds): 3.197382336
Average time elapsed for matrix-v2 (in seconds): 0.493222546
Speedup: 6.482636204

Mann-Whitney U test result:
U statistic = 10000.0
P-value = 1.2810718345817004e-34
The speedup is statistically significant (p < 0.05).
```

Computations in code `item3c.py`

Item 3d

```
Shiro@LAPTOP-U2KIJPRU MINGW64 ~/cs21lab08 (master)
$ python item3d.py
Average cache miss rate (in percent) for matrix-v1: 9.378729834
Average cache miss rate (in percent) for matrix-v2: 0.070648609
Improvement: 132.75179714468848

Mann-Whitney U test:
U statistic = 10000.0
P-value = 1.2810718345817004e-34
Change is statistically significant (p < 0.05)
```

Computations in code `item3d.py`

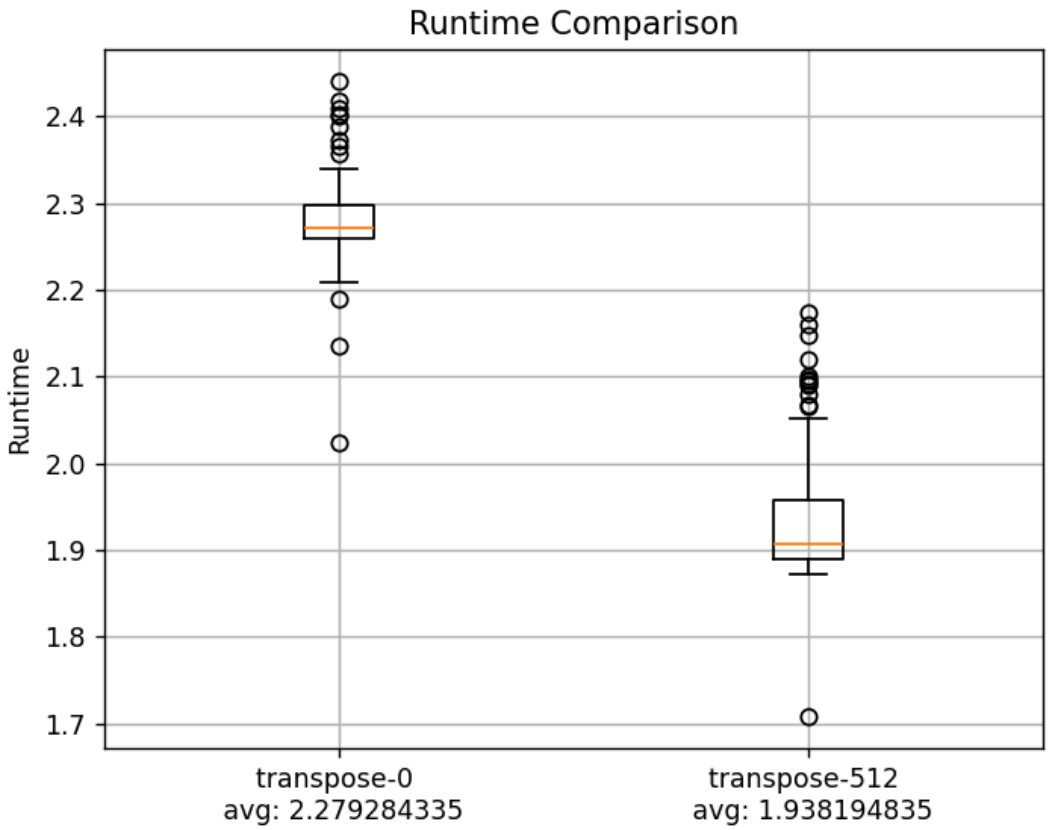
Item 4

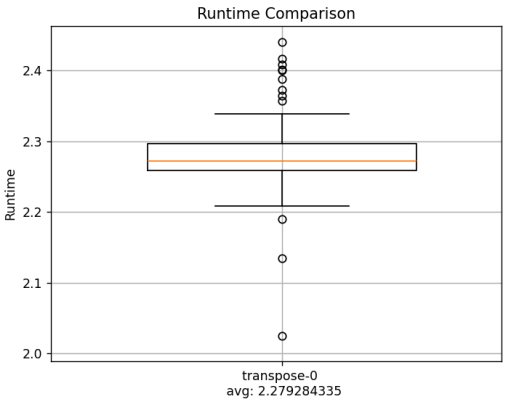
In the original nesting order, `temp` is written to by column and `data` is read by row. Swapping the nesting order will just make `temp` written to by row and `data` read by column. This simply just switches the way in which the arrays are written to or read, but doesn't necessarily improve the overall memory locality of the algorithm.

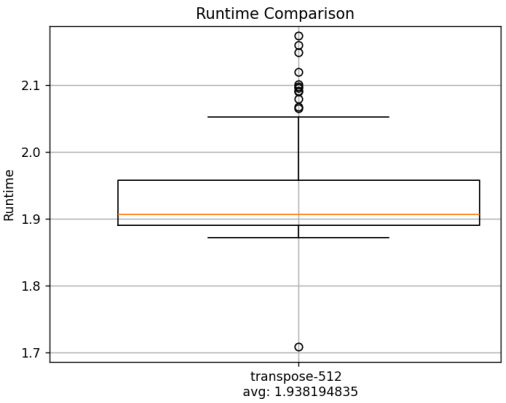
Item 5

For writing to `temp`, despite writing by column, the next index would be separated only by a `blocksize` number of rows. This improves the spatial locality when moving from column to column within a block. Compare this to the original, there you write by a whole column in the memory, so that when you need to move onto the next column, you have to go through all of the rows of the image.

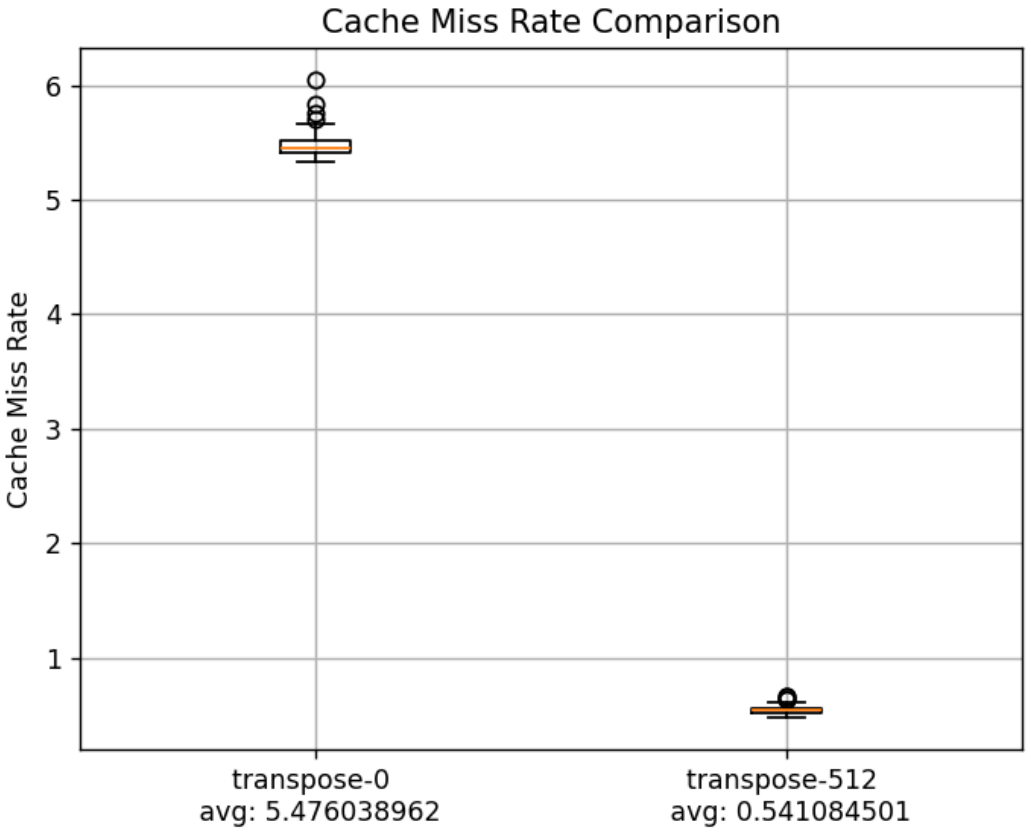
Item 6a

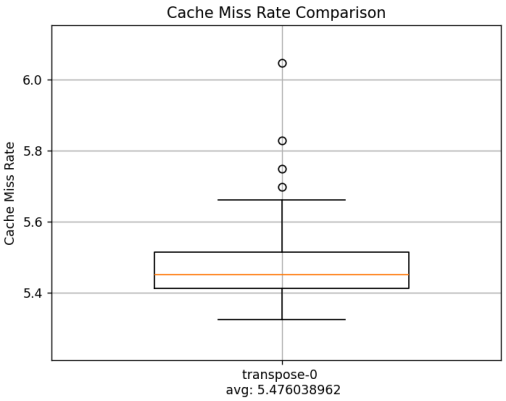

Runtime Comparison
Runtime
2.4
2.3
2.2
2.1
2.0
1.9
1.8
1.7
transpose-0
avg: 2.279284335
transpose-512
avg: 1.938194835

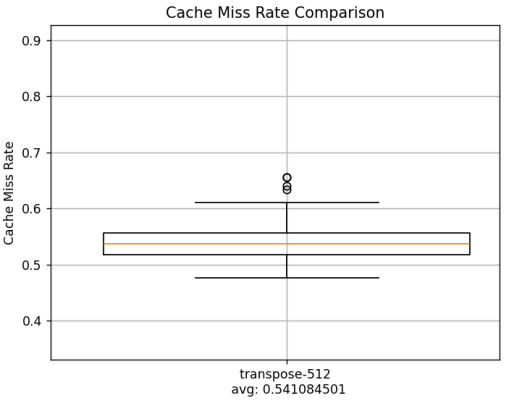

Runtime Comparison
Runtime
2.4
2.3
2.2
2.1
2.0
transpose-0
avg: 2.279284335


Runtime Comparison
Runtime
2.1
2.0
1.9
1.8
1.7
transpose-512
avg: 1.938194835

Item 6b


Cache Miss Rate Comparison
Cache Miss Rate
6
5
4
3
2
1
transpose-0
avg: 5.476038962
transpose-512
avg: 0.541084501


Cache Miss Rate Comparison
Cache Miss Rate
6.0
5.8
5.6
5.4
transpose-0
avg: 5.476038962


Cache Miss Rate Comparison
Cache Miss Rate
0.9
0.8
0.7
0.6
0.5
0.4
transpose-512
avg: 0.541084501

Item 6c

```
Shiro@LAPTOP-U2KIJPRU MINGW64 ~/cs21lab08 (master)
$ python item6c.py
Average time elapsed for transpose-0 (in seconds): 2.279284335
Average time elapsed for transpose-512 (in seconds): 1.938194835
Speedup: 1.175983082

Mann-Whitney U test result:
U statistic = 9980.0
P-value = 2.3334751143002393e-34
The speedup is statistically significant (p < 0.05).
```

Computations in code `item6c.py`

Item 6d

```
Shiro@LAPTOP-U2KIJPRU MINGW64 ~/cs21lab08 (master)
$ python item6d.py
Average cache miss rate (in percent) for transpose-0: 5.476038962
Average cache miss rate (in percent) for transpose-512: 0.541084501
Improvement: 10.120487554017606

Mann-Whitney U test:
U statistic = 10000.0
P-value = 1.2809996731450627e-34
Change is statistically significant (p < 0.05)
```

Computations in code `item6d.py`