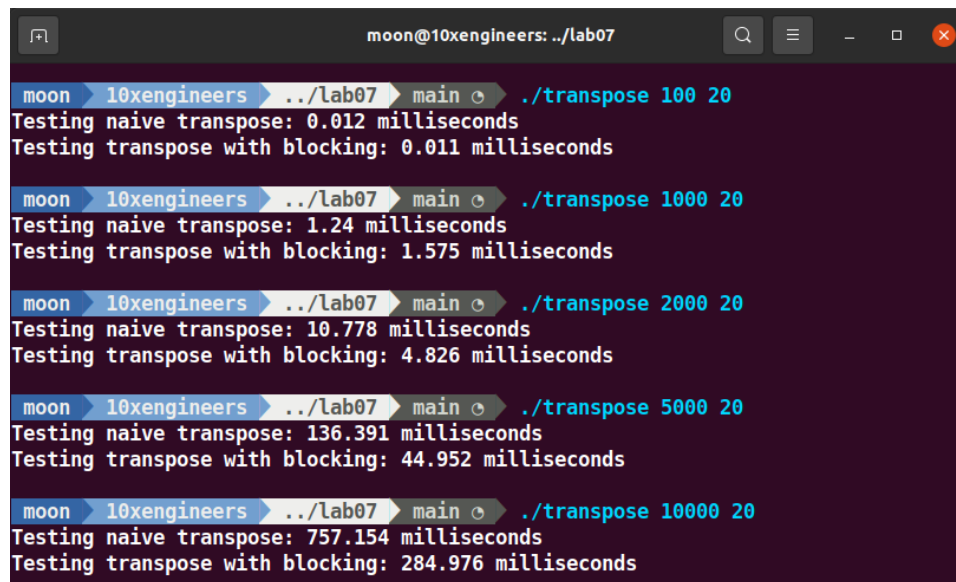


## Task 4

### Part 1 - Changing Array Sizes

**Question 1: At what point does cache blocked version of transpose become faster than the non-cache blocked version?**

Cache blocked version become faster than non-cache blocked version when  $n=2000$  i.e. the size of array is  $2000 \times 2000$



```
moon@10xengineers: ../lab07 main ➤ ./transpose 100 20
Testing naive transpose: 0.012 milliseconds
Testing transpose with blocking: 0.011 milliseconds

moon@10xengineers: ../lab07 main ➤ ./transpose 1000 20
Testing naive transpose: 1.24 milliseconds
Testing transpose with blocking: 1.575 milliseconds

moon@10xengineers: ../lab07 main ➤ ./transpose 2000 20
Testing naive transpose: 10.778 milliseconds
Testing transpose with blocking: 4.826 milliseconds

moon@10xengineers: ../lab07 main ➤ ./transpose 5000 20
Testing naive transpose: 136.391 milliseconds
Testing transpose with blocking: 44.952 milliseconds

moon@10xengineers: ../lab07 main ➤ ./transpose 10000 20
Testing naive transpose: 757.154 milliseconds
Testing transpose with blocking: 284.976 milliseconds
```

**Question 2: Why does cache blocking require the matrix to be a certain size before it outperforms the non-cache blocked code?**

When we use cache blocking, we divide the matrix into smaller blocks. But cache blocking also adds some extra work. We need to manage these blocks, which means there is additional overhead there. Because we had to keep track of which blocks are being processed and the movement of data in and out of cache.

With cache blocking, there is a higher chance that the data we need for the current block is not already in the cache, so it may result in more cache misses. Therefore, for smaller matrix sizes, the benefits of cache blocking, such as improved cache utilization, may not outweigh the overhead and increased cache misses.

.....

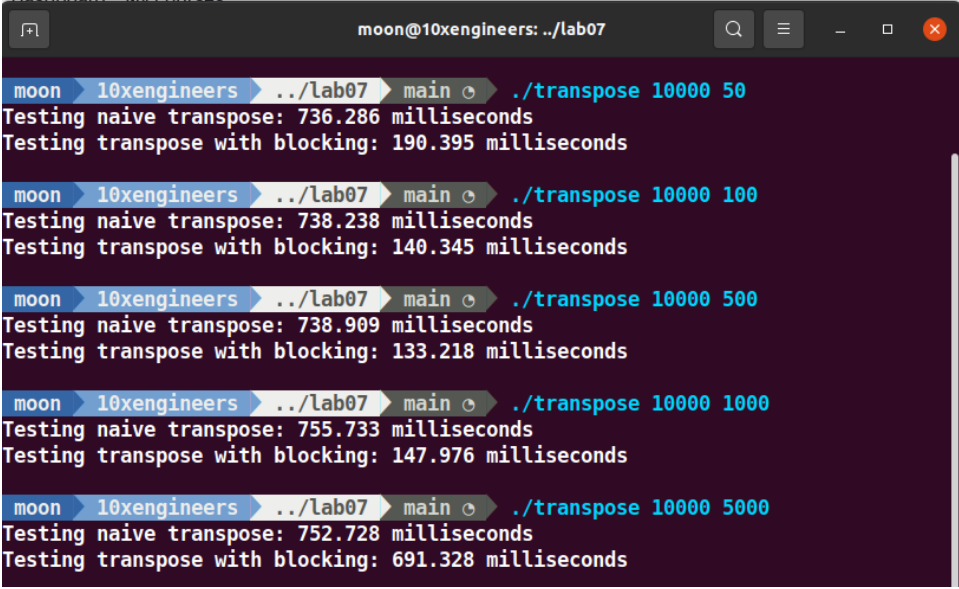
## Part 2 - Changing Block Size

**Question 3: How does performance change as blocksize increases? Why is this the case?**

The performance improves initially and then started decreasing.

The reason for the initial performance improvement with larger blocksize is related to cache utilization. With a larger block size, a higher number of matrix elements are processed together, which improves data locality and reduces cache misses. The cache can hold more elements from the same block, and the reuse of these elements within the block leads to better cache performance.

However, when the block size becomes too large, such as when it is set to 1000 or 5000, the performance starts to decrease. This happens because as the block size increases, the number of blocks or chunks decreases, and each block becomes larger. This larger block may exceed the cache capacity, resulting in more frequent cache evictions and increased cache misses. Consequently, the performance suffers due to the increased main memory accesses.



```
moon@10xengineers: ~/lab07
moon ➤ 10xengineers ➤ ../lab07 ➤ main ➤ ./transpose 10000 50
Testing naive transpose: 736.286 milliseconds
Testing transpose with blocking: 190.395 milliseconds

moon ➤ 10xengineers ➤ ../lab07 ➤ main ➤ ./transpose 10000 100
Testing naive transpose: 738.238 milliseconds
Testing transpose with blocking: 140.345 milliseconds

moon ➤ 10xengineers ➤ ../lab07 ➤ main ➤ ./transpose 10000 500
Testing naive transpose: 738.909 milliseconds
Testing transpose with blocking: 133.218 milliseconds

moon ➤ 10xengineers ➤ ../lab07 ➤ main ➤ ./transpose 10000 1000
Testing naive transpose: 755.733 milliseconds
Testing transpose with blocking: 147.976 milliseconds

moon ➤ 10xengineers ➤ ../lab07 ➤ main ➤ ./transpose 10000 5000
Testing naive transpose: 752.728 milliseconds
Testing transpose with blocking: 691.328 milliseconds
```

The terminal output shows the performance of a naive transpose algorithm versus a blocked transpose algorithm for a 10000x5000 matrix. The blocked algorithm consistently outperforms the naive algorithm, with the performance gap widening as the block size increases from 50 to 5000. The blocked algorithm's performance starts at 190.395 ms for a block size of 50 and reaches a minimum of 133.218 ms for a block size of 500, before increasing to 691.328 ms for a block size of 5000. The naive algorithm's performance remains relatively constant, ranging from approximately 736 to 756 milliseconds.

Code for the implemented transpose\_blocking is:

```
void transpose_blocking(int n, int blocksize, int *dst, int *src) {  
    // YOUR CODE HERE  
    int chunk_count = n / blocksize;  
    // normal case  
    // i: iterates over the chunk in rows  
    // j: iterates over the chunk in columns  
    // x: iterates over the elements within each row chunk  
    // y: iterates over the elements within each column chunk  
    for (int i = 0; i < chunk_count; i++) {  
        for (int j = 0; j < chunk_count; j++) {  
            for (int x = 0; x < blocksize; x++) {  
                for (int y = 0; y < blocksize; y++) {  
                    dst[y + j * blocksize + (x + i * blocksize) * n] = src[x + i * blocksize + (y + j * blocksize) * n];  
                }  
            }  
        }  
    }  
  
    // If n is not a multiple of blocksize, handle the edge case  
    int normalsize = chunk_count * blocksize;  
    for (int k = 0; k < n; k++) {  
        for (int z = normalsize; z < n; z++) {  
            dst[z + k * n] = src[k + z * n];  
            dst[k + z * n] = src[z + k * n];  
        }  
    }  
}
```