GitHub Repository

1 Problem Description

Develop an algorithm to transpose non-symmetric matrices of size $N \times N$, where N is a power of 2. Assess the algorithm's performance by measuring effective bandwidth under different compiler optimization options (-O0, -O1, -O2, and -O3). Analyze cache behavior to evaluate implementation efficiency.

1.1 Matrix definition

The matrix data is defined as a 1D array of size $N \times N$, where the element at position i, j is stored at the index $i \times N + j$. The matrix is stored in row-major order.

1.2 Matrix transposition[1]

Algorithm 1 Simple matrix transposition

```
function T0
    result ← Matrix shape.y × shape.x
    for i ← 0 to shape.x do
        for j ← 0 to shape.y do
            result[j][i] ← data[i][j]
        end for
    end for
    return result
end function
```

Algorithm 2 Block Matrix Transpose

```
function T2(block = 16)
    result ← Matrix shape.y × shape.x
    N ← shape.x
    M ← shape.y
    for ii ← 0 to N by block do
        for jj ← 0 to M by block do
            for i ← ii to min(ii + block, N) do
                for j ← jj to min(jj+block, M) do
                    result[j][i] ← data[i][j]
                end for
            end for
        end for
    end for
    return result
end function
```

The simple transposition algorithm (Algorithm 1) is straightforward and easy to implement. However, due to its implementation it is not cache-friendly, as it does not exploit the cache hierarchy. The block matrix transposition algorithm (Algorithm 2) divides the matrix into blocks of size $block \times block$ and transposes each block. This approach improves the cache behavior by reducing the number of cache misses.

[1] $matrix[i][j]$ notation is used instead of $matrix[i \times shape.y + j]$ due to space constraints and readability concerns.

1.3 Performance metrics

Time is registered using the `std::chrono` library. The effective bandwidth is calculated as the ratio between the number of bytes read and written and the time taken to perform the operation using the formula:

$$\text{eb} = \text{Effective bandwidth} = \frac{Br + Bw}{\text{t\_routine}} \quad (1)$$

$$= \frac{\text{size\_A} \times \text{size\_B} \times \text{dtype\_bytes}}{\text{time} \times 10^9} \quad (2)$$

2 Experimental setup

2.1 Hardware

The experiments were performed on a system equipped with a robust CPU cooler, ensuring that thermal throttling was effectively prevented. Table 1 and Table 2 detail the CPU and memory specifications, respectively.

Model	AMD Ryzen 5 5600X
Architecture	x86
Clock Speeds	3.7 GHz base, 4.6 GHz boost
Cache Levels	L1 384 KB, L2 3 MB, L3 32 MB
Cores, Threads	6, 12

Table 1: Processor specs.

Type	DDR4
Size	16 GB
Speed	3200 MHz
Memory Channels	Dual Channel
DOCP/AMP/XMP	DOCP 3200 MHz

Table 2: Memory specs.

2.2 Runs

To improve the accuracy of the measurements and reduce the OS noise, each experiment was repeated multiple times to have 1s run time, and the average value was calculated. The matrix size was varied from 2^1 to 2^{10}, and the block size was set to 16. All experiments were done with float data type. Results are presented in the next section.

3 Results

Performance Trends

Taking a look at Figure 1 we can notice that:

- No optimization exhibits the worst performance, which is expected as it applies no optimization techniques.
- O1 optimization shows a similar curve to no optimization but with slightly better performance. Both methods perform comparably.
- O2 and O3 optimizations demonstrate a more significant improvement globally.
- In O2 and O3 the simple method outperforms the block method for small matrices ($\leq 2^6$), while the block method performs better for larger matrices. This difference persists despite similar cache hit and miss rates for both methods. This may be due to overhead of block management.

Cache Comparison

Comparing cache characteristics between O3 simple and block optimizations from Figure 3, 4, and 5:

- Both methods exhibit similar instruction, data, and unified cache behavior.
- However, there's a noticeable difference in cache misses, particularly in the D1 miss rate, with O3 simple at 3.3% and O3 block at 1.3% when $N = 2^8$.

Decrease in Cache References

From Figure 6 we can observe that as optimization levels increase, cache references decrease. This reduction is attributed to more efficient code resulting from higher optimization levels, leading to fewer memory accesses. This decrease is evident in D references. Note that the D1 miss rate increases as the matrix size increases and as the optimization level increases. This increase is primarily in the write cache, suggesting that prefetching may be beneficial.

Similar Cache Miss

For matrices with size $\leq 10^6$, all optimization levels exhibit similar cache miss behavior. However, differences become more pronounced for matrices with size $> 10^7$.

Performance Variation between Simple and Block Methods

The performance difference between the simple and block methods stems from various factors:

- Cache efficiency: Block matrix transpose benefits from better cache utilization due to smaller block sizes, particularly noticeable for large matrices. However, this advantage diminishes for smaller matrices due to overhead.
- Data movement: Sequential transpose accesses matrix elements sequentially, leading to poor spatial locality for large matrices and longer access times. Block matrix transpose operates on smaller blocks, reducing the distance between accessed elements and improving data movement efficiency.
- Cache locality: The nested loop sequential transpose may underutilize the cache for large matrices, leading to increased cache misses and slower memory access. In contrast, block matrix transpose operates on smaller submatrices that fit better into the cache, resulting in improved cache locality and reduced cache misses. However, for this assignment, with the largest matrix being 2^{10} (about 3MB), this difference may not be significant.

Overall, the difference in miss rates between the simple and block methods for larger matrices is a combination of cache line utilization, spatial locality, and loop optimizations.

4 Future work

The most straightforward method to speed up the transposition process is to parallelize the algorithm. This can be achieved using a block method, where each block is transposed in parallel. This approach can be implemented using OpenMP or CUDA for CPUs and GPUs, respectively. Since the transpose operation operates on a distinct matrix, the algorithm can directly manipulate memory without necessitating the use of semaphores or mutexes.

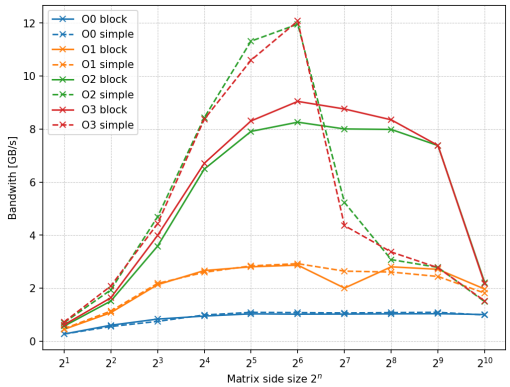

Figure 1: Effective bandwidth (float)

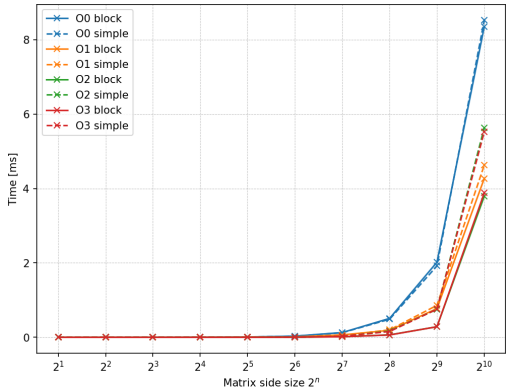

Figure 2: Execution time (float)

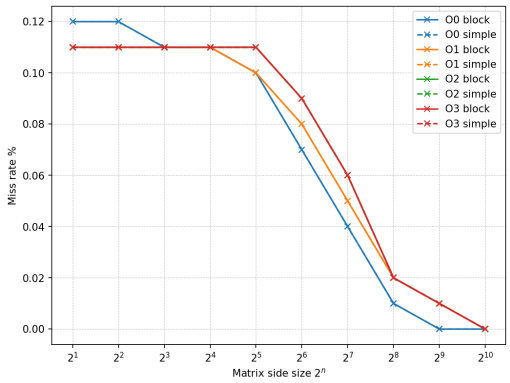

Figure 3: L1 miss rate

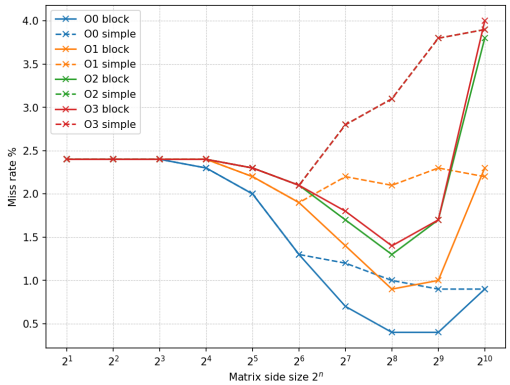

Figure 4: D1 miss rate

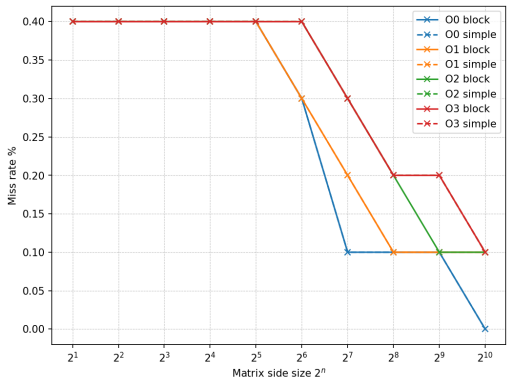

Figure 5: LL miss rate

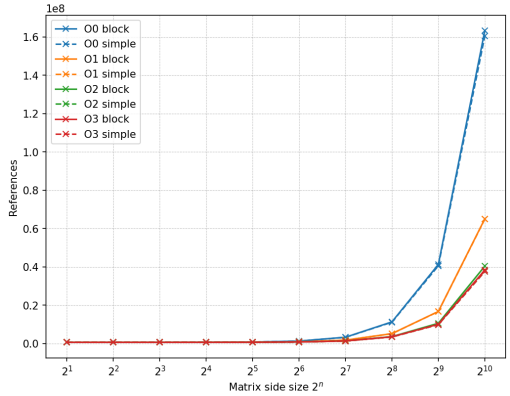

Figure 6: D references