

CMOR 521 HW1

Swarnamoy Ghosh

1 RUNNING THE CODE

This section briefly explains how to run the code for this assignment. In this assignment, we implement naive, cache-blocked, and recursive versions of a matrix transposition function.

1. As instructed, the driver files are in the main directory - **main.c**, **makefile**, **matmul_blocked.cpp** and **matmul_recursive.cpp**.
2. **mat_trans.c** inside **src** sub-directory.
3. A header file - **transpose.h** inside the **include** sub-directory.

The script **mat_trans.c** contains the necessary functions for different matrix transposition algorithms.

To check the implementation of the function 'time_transpose', go to the home directory and run the following commands in the terminal -

1. **\$ make**
2. **\$./transpose n**

Here, the 'n' is the size of n x n matrix. We can change the number of trials by going inside 'mat_trans.c' and changing the variable 'trials' to a desired number. By default, the program prints the minimum runtime over 25 trials and the maximum relative error compared to a reference implementation.

To check the timings of cached block method for different block sizes and return an optimal block size, run the executable 'transpose' as -

\$./transpose --sweep-block n trials For example for a 1024 x 1024 matrix and with 30 trials, do -

\$./transpose --sweep-block 1024 30

Similarly, for the recursive transpose, to get the timings and the optimal threshold, do -

\$./transpose --sweep-thresh 1024 30

That is all we need to know!

2 ANALYSIS - TRANPOSE

1. For the cache-blocked transpose, report timings for different block sizes and determine an optimal block size.-

```
(base) swarnamoyghosh@Swarnamoys-MacBook-Pro CMOR_HW1 % ./transpose --sweep-block 2048 40
impl,n,param,time,err
blocked,2048,4,7.1269999899e-03,0.000e+00
blocked,2048,8,6.1790000182e-03,0.000e+00
blocked,2048,12,6.3419999788e-03,0.000e+00
blocked,2048,13,7.8330000106e-03,0.000e+00
blocked,2048,14,9.4939999981e-03,0.000e+00
blocked,2048,15,1.2031999999e-02,0.000e+00
blocked,2048,16,1.2683000008e-02,0.000e+00
blocked,2048,24,1.6449000017e-02,0.000e+00
blocked,2048,32,1.3815000013e-02,0.000e+00
blocked,2048,48,1.7131000001e-02,0.000e+00
```

```

blocked,2048,64,1.3896000019e-02,0.000e+00
blocked,2048,96,1.7407999985e-02,0.000e+00
blocked,2048,128,1.7355999997e-02,0.000e+00
blocked,2048,192,1.9697000011e-02,0.000e+00
blocked,2048,256,1.9870000018e-02,0.000e+00
Best block size for n=2048: Block=8 (time=6.179000e-03 s)

```

Therefore, the best block size is **8**.

2. For the recursive transpose, instead of terminating the recursion at $n=1$, terminate by running a "microkernel" if the matrix is smaller than some threshold size. Report timings for different threshold sizes, and determine an optimal threshold size.

```

(base) swarnamoyghosh@Swarnamoys-MacBook-Pro CMOR_HW1 % ./transpose --sweep-thresh 2048 40
impl,n,param,time,err
recursive,2048,8,1.4521999983e-02,0.000e+00
recursive,2048,16,1.3406999991e-02,0.000e+00
recursive,2048,24,1.3374000002e-02,0.000e+00
recursive,2048,32,1.3737000001e-02,0.000e+00
recursive,2048,48,1.3739000016e-02,0.000e+00
recursive,2048,64,1.3829999982e-02,0.000e+00
recursive,2048,96,1.3795000006e-02,0.000e+00
recursive,2048,128,1.7454999994e-02,0.000e+00
recursive,2048,192,1.7396999989e-02,0.000e+00
recursive,2048,256,1.9688999979e-02,0.000e+00
recursive,2048,384,1.9710000022e-02,0.000e+00
recursive,2048,512,1.9946999993e-02,0.000e+00
Best threshold for n=2048: Thresh=24 (time=1.337400e-02 s)

```

The best threshold size is **24**.

3. Plot runtimes for each implementation (use optimal block/threshold sizes) for matrix sizes and discuss what you observe.

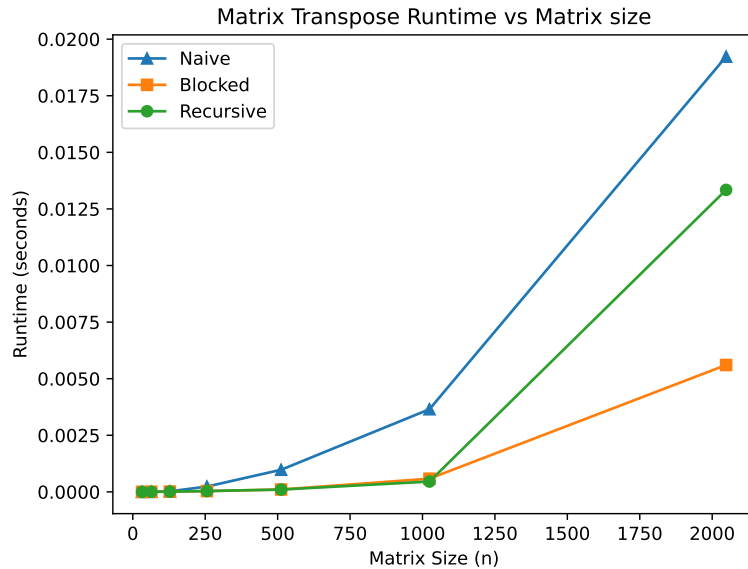


Figure 1: Runtimes for Naive, blocked and recursive transpose algorithms for different matrix size n . The optimal block size and threshold size chosen to be 8 and 24 respectively.

In Fig. 1, the naive transpose performs comparably to blocked and recursive for small matrix size n , since, the working sets fits well in cache and the runtime is dominated by loop overhead. Blocked and recursive,

in this case doesn't utilize the cache benefits. As n increases, naive method performs slower due to poor cache behaviour caused by strided access (AT is stored in row-major order). The cache-block transpose works well for large n because it minimizes the cache-misses by creating sub-blocks of A and AT and keeping them in cache. The recursive method improves too at large n relative to naive but remains slower than blocked, likely due to recursion overheads.

4. **How many slow memory reads/writes are required for naive matrix transposition, assuming that the full matrix is large enough so that it does not fit into fast memory?**

Assuming the matrix is sufficiently large so that it does not fit into fast memory, i.e. no effective data reuse occurs, then each element of A must be read and written n^2 times.

5. **If AT is stored in column-major format, which matrix transposition algorithm will be the fastest?**

If AT is stored in column-major format, then naive-transpose is typically fastest, since it fixes the large stride problem, and both arrays are accessed sequentially.

Blocking or recursive will in turn add extra loop overheads, complicated overflow with no locality improvement.

3 ANALYSIS - MATRIX MULTIPLICATION

For this part, I have reused the source cpp files already provided to us on Canvas. The recursive matrix-multiplication is implemented using the 'matmul_recursive.cpp', where I modified it a bit to return the optimum runtime over 25 trials, and checking the relative error.

To run matmul_recursive.cpp, type the following at the home directory -

```
$ g++ -O3 -march=native -std=c++17 matmul_recursive.cpp -o matmul
$ ./matmul n
```

where n is the matrix size.

For example, testing for $n = 1024$,

```
./matmul 1024
Matrix size n = 1024, recursive threshold = 8
max relative error = 0
PASSED correctness check.
Elapsed time for recursive matmul in (secs): 1.43596
```

I swept the threshold $t \in \{2, 4, 8, 16, 32, 64, 128\}$ for matrix of size $n = 1024$, and found $t = 4$ as optimum.

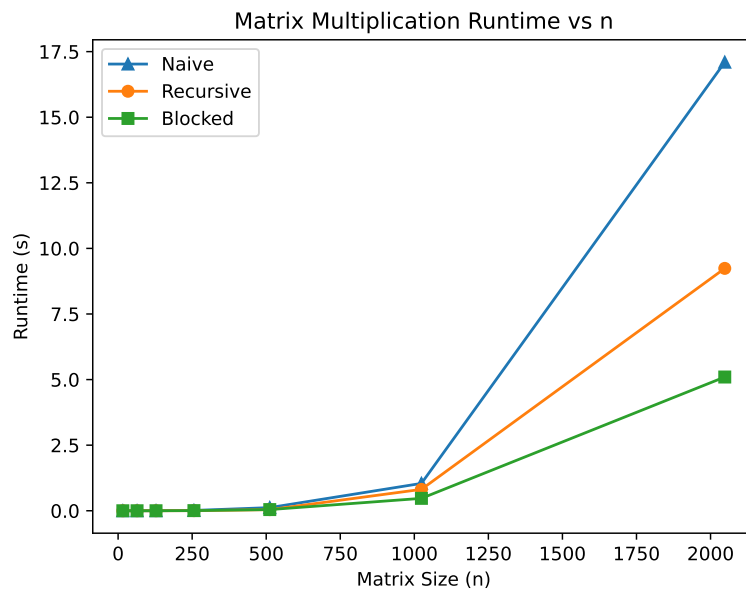


Figure 2: Runtimes for Naive, blocked and recursive matmul algorithms for different matrix size n . The optimal block size and threshold size chosen to be 8 and 4 respectively.

Fig 2. shows the runtime of naive, recursive, and cache-blocked matrix multiplication as a function of matrix size.

For small matrices, all three methods perform similarly because the working set fits entirely in cache and execution is dominated by loop overhead. As matrix size increases, the naive implementation becomes significantly slower due to poor cache locality and increased memory traffic.

The recursive implementation improves performance relative to naive by improving locality through divide-and-conquer decomposition. However, its performance remains inferior to the blocked implementation for large matrices due to recursion overhead and the relatively small microkernel size.

At the largest tested size, the blocked algorithm achieved a runtime of 5.09746 seconds, compared to 9.23824 seconds for recursive and 17.0965 seconds for naive. This corresponds to speedups of approximately $3.35\times$ over naive and $1.81\times$ over recursive. These results demonstrate that explicit cache blocking provides the best scalability and cache efficiency for large matrix sizes.