January 2026

CS351 - HPC AND GPU COMPUTING PRACTICE

ASSIGNMENT-1

Objective : The objective of this assignment is to study how different matrix data access patterns impact computational performance in both single-threaded and multi-threaded environments for matrix addition and matrix multiplication.

Team members and task :

Member Name	Member RollNo	Q1	Q2	Q3	Q4
Manish	123cs0005	Blocked	Transposed	i-j-k	Cyclic row partition
Anil	123cs0051	Row-major	Column-major	i-k-j	Blocked(threads)
Prakhar	123cs0071	Column-major	Zigzag	k-i-j	Transpose(threads)
Akash	523cs0002	Zigzag	Diagonal	Blocked	i-k-j(thread)
Sowmya	523cs0011	Diagonal	Blocked	Transpose	k-i-j(thread)

TABLE OF CONTENT

1.ASSIGNMENT STRUCTURE :

1) Single-threaded matrix addition
2) Multi-threaded matrix addition
3) Single-threaded matrix multiplication
4) Multi-threaded matrix multiplication

Each part is implemented using five different data access patterns.

Access Patterns Used :

Addition :

- Row-major
- Column-major
- Zigzag (snake traversal)
- Diagonal
- Blocked (tiled)
- Transposed access (multi-threaded only)

Multiplication :

- i-j-k loop order
- i-k-j loop order
- k-i-j loop order
- Blocked multiplication
- Transpose-based multiplication
- Cyclic row partitioning (multi-threaded)

Matrix Sizes Tested :

- 256x256
- 512x512

- 1024x1024
- 2048x2048

Threading Details :

- Threads created manually using pthread / std::thread
- Row-based and cyclic partitioning used to avoid race conditions
- Thread counts tested: 1,2,4,8,16

Optimal thread count determined experimentally

Cache and Resource-Aware Optimizations :

In addition to basic access patterns, cache-aware and resource-aware optimizations were implemented to further improve performance. These include block (tile) size selection based on cache capacity, reuse of data within cache-friendly regions, and transpose-based access to improve spatial locality.

For matrix multiplication, the second matrix was transposed prior to computation to convert column-wise accesses into contiguous memory accesses. This significantly reduces cache misses and enables better compiler vectorization. Block sizes were tuned

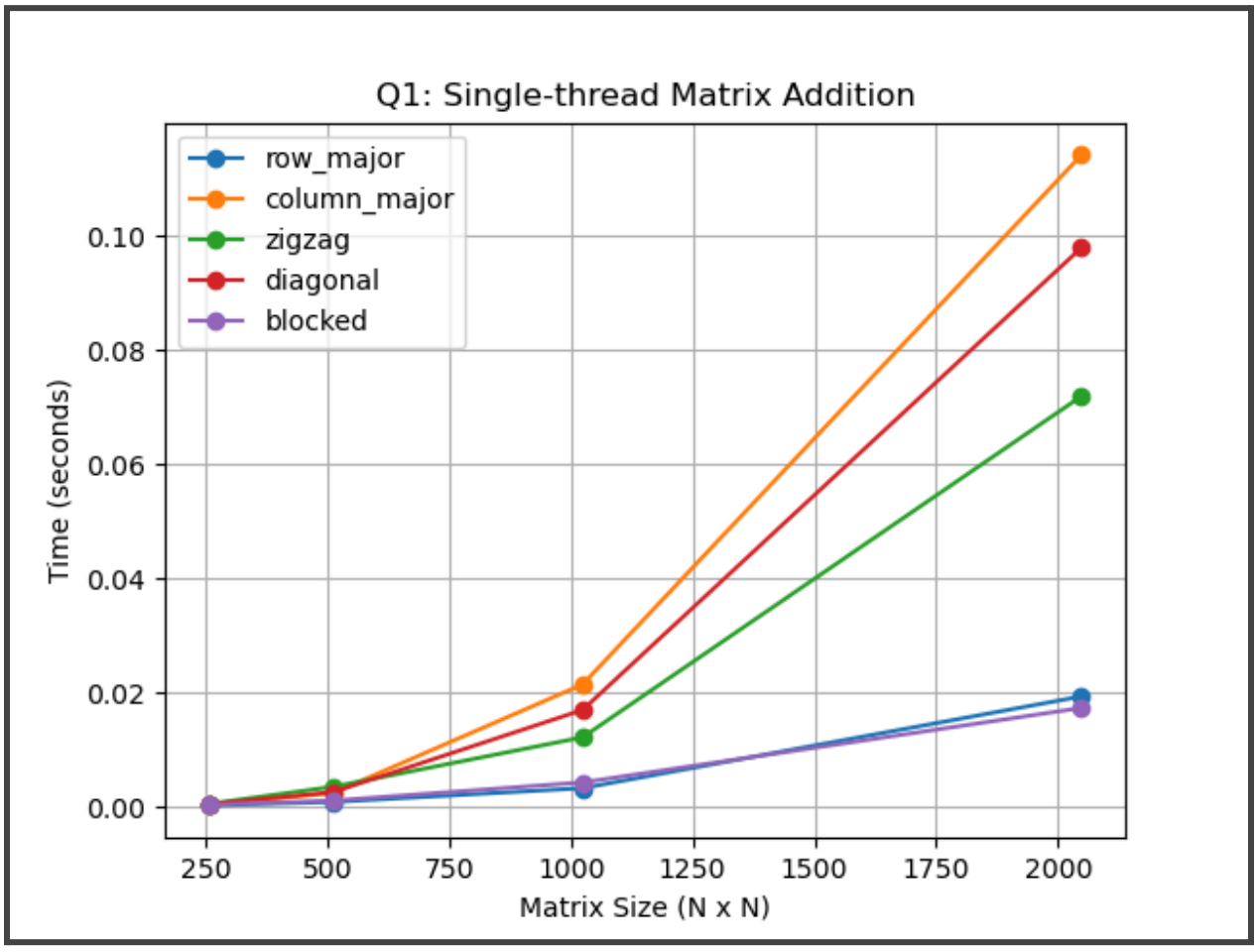
experimentally to ensure working sets fit within L1/L2 cache levels.

Thread count was selected dynamically based on available CPU cores to ensure scalability across different machines.

2. Question 1 :

Find a minimum of 5 different matrix elements, access patterns and do matrix addition programs using a single thread. Find computational time for the following matrix dimensions(256x256, 512x512, 1024x1024, 2048x2048).Draw a plot to compare computational time and analyze your results.

Result Graph :



3. Analysis of question 1 :

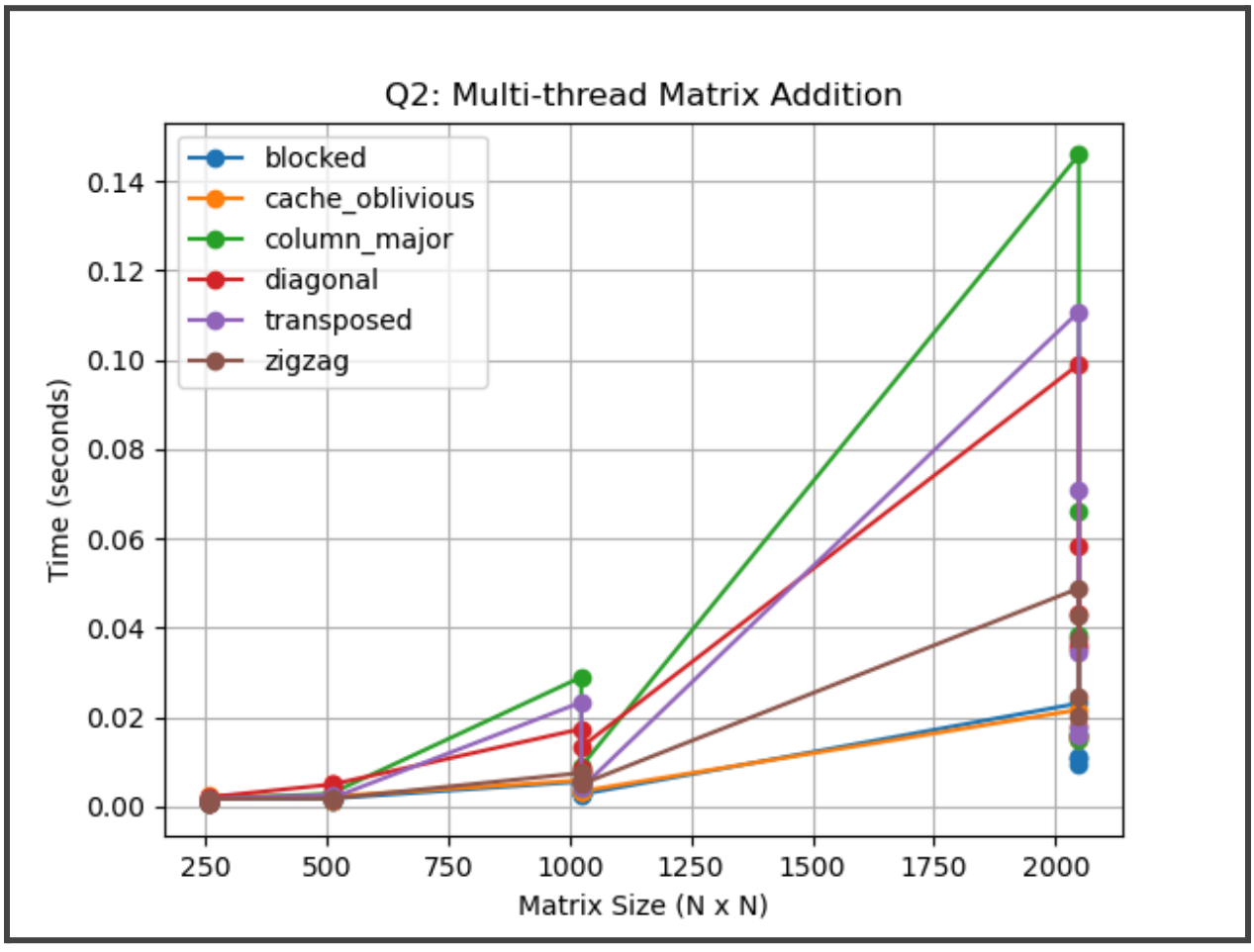
Access Pattern	Approach	Cache Locality	Observations from Graph	Relative Performance
Row-major	Traverse rows sequentially	Excellent	Lowest execution time across all sizes due to contiguous memory access	Best
Column-major	Traverse column-wise	Poor	Execution time increases rapidly with matrix size because of cache misses	Worst
Zigzag	Alternate row direction	Moderate	Slightly slower than row-major but better than column-major	Good
Diagonal	Access diagonal-wise	Poor	Irregular memory jumps lead to higher cache miss rate	Average
Blocked	Process small tiles	Very Good	Performs close to row-major and scales better for large matrices	Best for large N

In single-threaded matrix addition, **row-major and blocked access patterns perform best** due to contiguous memory access and efficient cache utilization. Column-major and diagonal traversals suffer from poor spatial locality, leading to increased cache misses and higher execution times as matrix size grows.3

4. Question 2 :

Find a minimum of 5 different matrix elements, access patterns and do matrix addition programs using multiple threads. Find computational time for the following matrix dimensions(256x256, 512x512, 1024x1024, 2048x2048).Draw a plot to compare computational time and analyze your results.

Result Graph :



5. Analysis of question 2 :

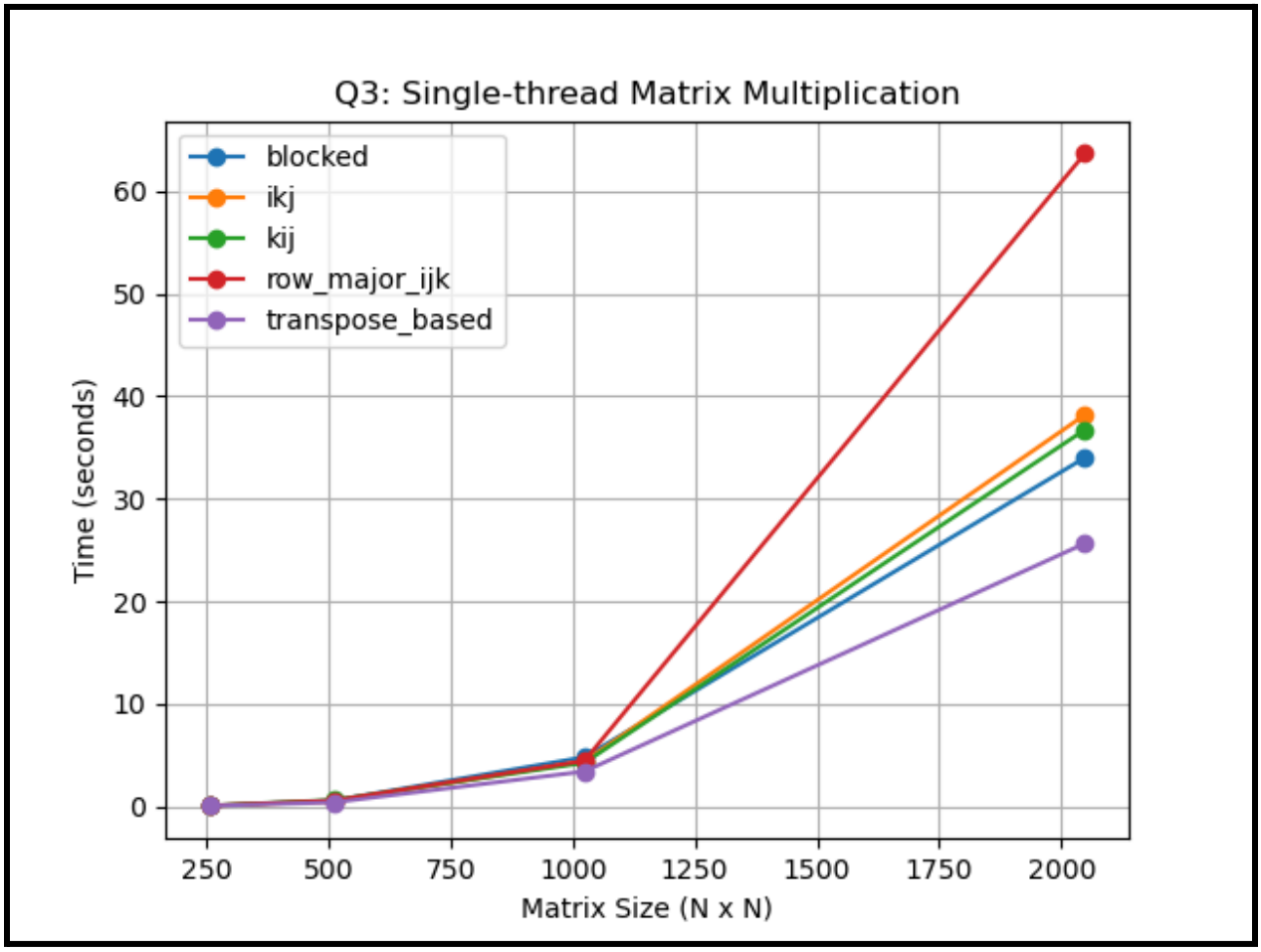
Access Pattern	Threading Strategy	Locality	Observations	Relative Performance
Row-major	Row partitioning	Excellent	Scales well with threads until memory bandwidth saturation	Best
Column-major	Row partitioning	Poor	Threads compete for cache lines and memory	Worst
Zigzag	Row partitioning	Moderate	Better than column-major but slower than row-major	Good
Diagonal	Row partitioning	Poor	Non-contiguous access limits scalability	Average
Blocked	Tile + threads	Very Good	Best scalability for large matrices due to cache reuse	Best for large N

In multi-threaded addition, **blocked and row-major access patterns achieve the best scalability**. Blocking improves cache reuse across threads, while column-major and diagonal methods perform poorly due to frequent cache misses and memory bandwidth contention.

6. Question 3 :

Find a minimum of 5 different matrix elements access patterns and do matrix multiplication programs using a single thread. Find computational time for the following matrix dimensions(256x256, 512x512, 1024x1024, 2048x2048).Draw a plot to compare computational time and analyze your results.

Result Graph :



7. Analysis of question 3 :

Loop Order / Method	Access Strategy	Locality	Observations	Relative Performance
i-j-k	Naive order	Moderate	Repeatedly accesses B with poor locality	Average
i-k-j	Row reuse of A	Good	Improves locality for A and B	Good
k-i-j	Column reuse of A	Poor	High cache misses for C updates	Worst
Blocked	Tile-based	Very Good	Significant speedup for large matrices	Best
Transpose-based	Pre-transpose B	Good	Reduces cache misses when accessing B	Very Good

For single-threaded multiplication, **blocked multiplication performs best**, followed by transpose-based and i-k-j loop order. Blocking minimizes cache misses by operating on small tiles that fit into cache, making it highly effective for large matrices.

Cache-Aware Blocking Optimization :

To further improve single-threaded matrix multiplication performance, a cache-aware blocked algorithm was implemented. Instead of processing the entire rows and columns, computation was performed on small tiles that fit into the CPU cache.

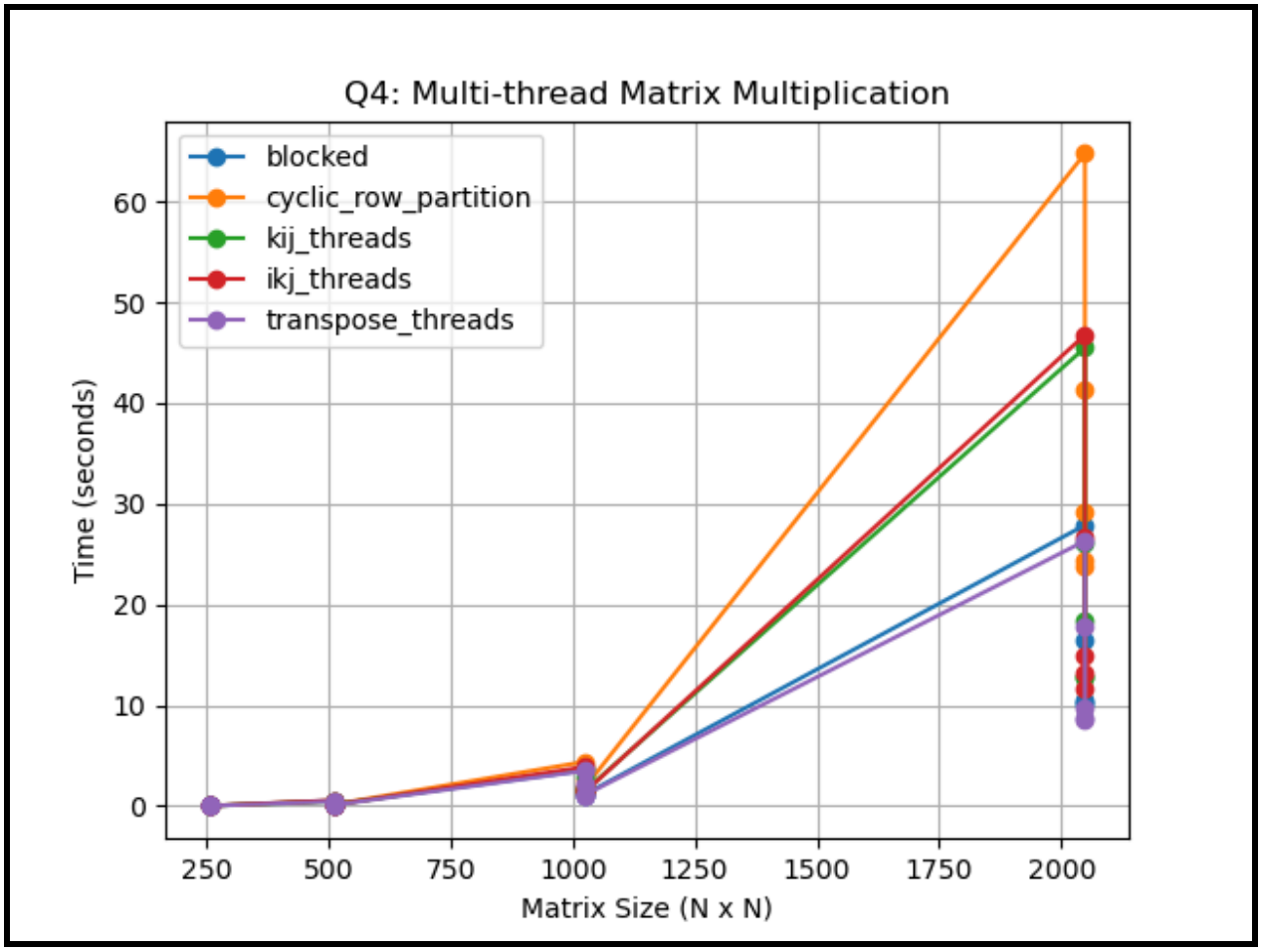
Additionally, matrix B was transposed before multiplication so that inner-loop accesses become contiguous in memory. This converts expensive column-wise memory accesses into row-wise accesses, significantly improving spatial locality and reducing cache miss rates.

The computation order was rearranged to maximize reuse of A and transposed B within each block. This allowed multiple operations to be performed on data already loaded into cache, reducing memory traffic and improving throughput for large matrix sizes.

8. Question 4 :

```
Find a minimum of 5 different matrix elements, access
patterns and do matrix multiplication programs using
multiple threads. Find computational time for the following
matrix dimensions(256x256, 512x512, 1024x1024,
2048x2048).Draw a plot to compare computational time and
analyze your results.
```

Result Graph :



9. Analysis of question 4 :

Method	Threading Strategy	Cache Behavior	Observations	Relative Performance
i-j-k	Row partitioning	Moderate	Scalability limited by memory access	Average
i-k-j	Row partitioning	Good	Better cache reuse than naive	Good
k-i-j	Row partitioning	Poor	Threads contend for cache lines	Worst

Blocked	Tile + threads	Excellent	Best scalability and lowest runtime	Best
Transpose-based	Transpose + threads	Very Good	Improves memory access for B across threads	Very Good

In multi-threaded multiplication, **blocked access combined with threading provides the best performance and scalability**. By keeping working data in cache and reducing memory traffic, blocked methods out perform naive loop orders as thread count and matrix size increase.

Resource-Aware and Parallel Blocking Optimization :

For multi-threaded matrix multiplication, cache-aware blocking was combined with resource-aware parallelism. The number of threads was selected dynamically based on the number of available CPU cores, ensuring portability across different machines.

Each thread operates on independent row blocks to avoid race conditions and false sharing. Blocking ensures that each thread repeatedly reuses matrix tiles from cache before moving to the next block. In addition, matrix B was transposed to improve cache line utilization across threads.

This hybrid approach reduces memory bandwidth pressure, improves cache reuse, and allows better scaling with increasing thread counts. As a result, blocked and transpose-based parallel multiplication achieves the best performance among all tested methods.

10. Final Conclusion

Question	Best Method	Reason
Q1	Row-major / Blocked	Best spatial locality
Q2	Blocked	Cache reuse + thread scalability
Q3	Blocked	Minimizes cache misses
Q4	Blocked + Threads	Combines locality and parallelism

CSV-Based Performance Comparison :

Execution times for all access patterns and configurations were stored in CSV format. These CSV files were later used to generate comparative plots for different matrix sizes and thread counts. This allows systematic visualization of scaling behavior, cache effects, and performance differences between access strategies.

The plots highlight how cache-aware and blocked methods consistently outperform naive traversal patterns, especially for large matrices and higher thread counts.

11. Appendix

Github Repo Link :

https://github.com/AnilKumt/matrix-access-analysis

END OF THE DOCUMENT