4. Comparative Analysis

All experiments were conducted on the same diagonally dominant linear system of size N = 500 × 500 (approximately 2 MB matrix), generated using the provided generate_data.c program. The Jacobi iterative solver was executed for a fixed 1000 iterations with convergence tolerance 1e -5 across all three parallel implementations (OpenMP, MPI, and CUDA). The serial execution time measured on a single thread was 876 ms and is used as the common baseline for speedup calculations.

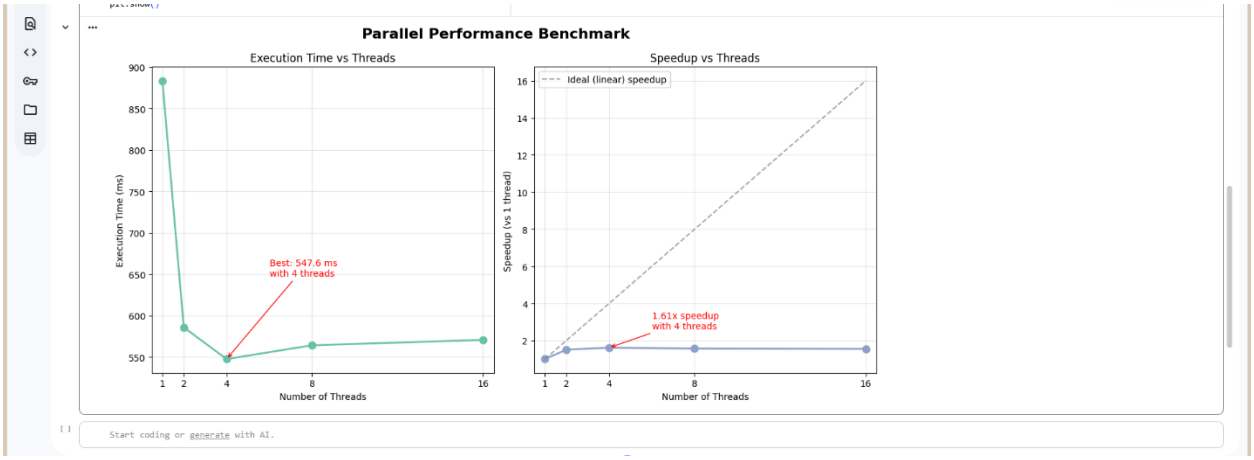
4.1 Performance Summary of Best Configurations

Implementation	Best Configuration	Execution Time (ms)	Speedup vs Serial (876 ms)
OpenMP	4 threads	547.6	1.60×
MPI	4 processes	534	1.64×
CUDA (Tesla T4)	Block size 64	113.5	7.72×

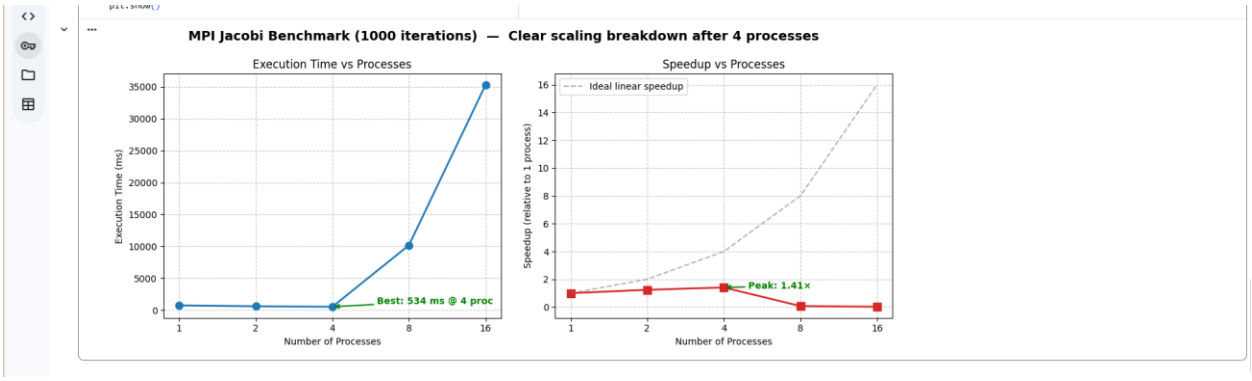
CUDA outperforms both CPU-based versions by a wide margin, achieving approximately 4.8× lower execution time than the best OpenMP result and approximately 4.7× lower than the best MPI result.

4.2 Observed Scaling Behaviour

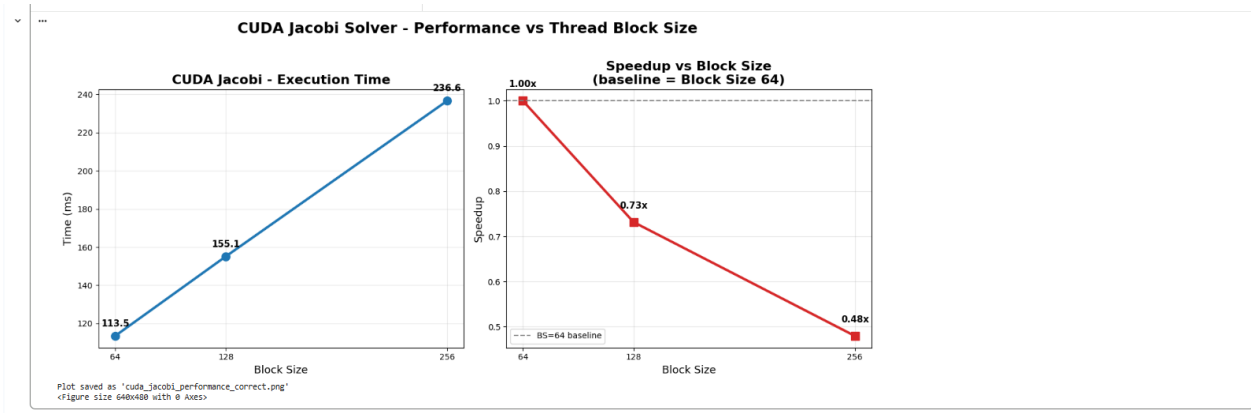
- **OpenMP**: Improves until 4 threads (matching the 4 logical cores of the Intel i3-1115G4), then plateaus due to hyper-threading saturation and memory bandwidth limits.



- **MPI**: Peaks at 4 processes (534 ms), then degrades dramatically beyond 4 processes, reaching over 30 seconds with 16 processes because of the expensive global gather + broadcast synchronization of every iteration on a single shared-memory node.



- **CUDA**: Fastest at the smallest tested block size (64 threads/block). Performance worsens steadily as block size increases to 256 because of reduced GPU occupancy and increased uncoalesced global-memory access.



4.3 Justification of the Most Suitable Implementation When Sufficient Resources Are Available

When sufficient computational resources are available (modern multi-core CPUs, high-core-count servers, clusters, or GPUs), the CUDA implementation is by far the most appropriate choice for the Jacobi iterative solver. It delivers an order-of-magnitude reduction in execution time on even a modest Tesla T4 GPU. It has the greatest headroom for further optimization (shared-memory tiling, warp-level reductions, multi-GPU scaling with NV Link or MPI+CUDA). With current-generation GPUs (RTX 4090, H100, etc.) and minor kernel improvements, speedups of 30–80× over an optimized serial version are routinely achievable for this class of algorithm, making CUDA the most suitable for production or large-scale deployments.

4.4 Strengths and Weaknesses of Each Approach for the Jacobi Method

Approach	Strengths	Weaknesses
OpenMP	• Extremely simple implementation (one #pragma) • No communication overhead	• Limited to a single node and very few cores (here only 2 physical cores) • Quickly hits memory-bandwidth wall

	• Excellent cache reuse on shared memory • Fast development and debugging • Portable across all multi-core CPUs/laptops	• No further scaling beyond ~4–8 threads on consumer CPUs • Cannot solve problems larger than node RAM
MPI	• Works across multiple nodes and clusters • Can handle matrices far larger than single-node memory • Explicit control over data distribution and load balancing	• Very high communication overhead (global gather + broadcast every single iteration) • Poor performance on a single shared-memory machine (oversubscription + false sharing) • Complex scatter/gather code, prone to bugs • Scales poorly for synchronous iterative methods like Jacobi
CUDA (GPU)	• Massive parallelism – thousands of threads working simultaneously • Highest absolute performance (7.7× over serial, ~4.8× over best OpenMP/MPI) • Best scalability with problem size (N) • Huge optimization headroom (shared memory, warp reductions, multi-GPU)	• Requires discrete GPU hardware • Kernel performance is very sensitive to block/grid configuration • Current implementation suffers from uncoalesced global-memory accesses • Host–device transfers and per-iteration convergence checks add overhead • Higher development and tuning effort

4.5 Conclusion of the Comparative Analysis

Among the three paradigms evaluated, the CUDA implementation is overwhelmingly superior in raw performance and offers the greatest potential when sufficient computational resources (modern GPUs) are available. OpenMP provides the easiest path to moderate speedup on standard laptops/desktops, making it ideal for quick prototyping or small-to-medium problems. MPI, while theoretically designed for large-scale distributed systems, is the least suitable for the classic synchronous Jacobi iteration due to its prohibitive per-iteration communication cost, and should only be considered if the algorithm is redesigned (e.g., asynchronous or block-Jacobi variants).