Matrix Multiplication in Theory and Practice Math, Concurrency, and High-Performance Implementation (Final)

Prepared by: Somesh Diwan

September 16, 2025

Contents

1 Overview

This document collects theory, practical algorithms, and production-grade parallel implementation guidance for matrix multiplication. It includes:

- concise mathematical foundations and worked examples,
- complexity analysis and practical optimizations (tiling, Strassen),
- detailed Java concurrency patterns (ExecutorService, CyclicBarrier, Phaser, ForkJoin),
- verification, benchmarking, memory-layout, and deployment notes.

2 Mathematical background (concise)

2.1 Definition and rationale

Let $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{m\times p}$. The product $C = AB$ is the $n \times p$ matrix with entries

$$C_{ij} = \sum_{k=1}^{m} A_{ik}\, B_{kj}, \qquad 1 \le i \le n,\ 1 \le j \le p.$$

This corresponds to composition of linear maps and the row–column (dot-product) view.

2.2 Worked example

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} \implies C = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}.$$

2.3 Complexity recap

Naive arithmetic cost for $A \in \mathbb{R}^{n\times m}$ and $B \in \mathbb{R}^{m\times p}$:

$$T(n, m, p) = \Theta(nmp).$$

For square $N \times N$ matrices, $T(N) = \Theta(N^3)$. Practical work focuses on lowering constants (tiling, vectorization, BLAS) or asymptotics (Strassen and theoretical research).

3 Algorithms and practical optimizations

3.1 Blocking / tiling

Partition matrices into $b \times b$ tiles and compute block multiplications

$$C_{(i,j)} \mathrel{+}= A_{(i,k)} \cdot B_{(k,j)}.$$

Choose b to maximize reuse in L1/L2 caches — tune empirically per hardware.

3.2 Strassen (2x2 block sketch)

Strassen reduces the number of half-size multiplications from 8 to 7, giving exponent $\log_2 7 \approx 2.807$. For two-by-two block matrices A, B, compute 7 M matrices (see main text) and combine them to form C. Strassen trades multiplications for additions and is worthwhile only for large enough matrices due to overhead and numerical care.

4 Production-grade concurrent Java: implementation and explanation

Below is a robust, production-minded Java example using flat arrays, row-block partitioning, and two `CyclicBarrier` phases. Compile and run in a modern JDK.

```java
/* MatrixMultiplyParallel.java */
import java.util.*;
import java.util.concurrent.*;

public class MatrixMultiplyParallel {

    public static void main(String[] args) throws Exception {
        final int N = 600; // rows of A and C
        final int M = 500; // cols of A, rows of B
        final int P = 400; // cols of B and C
        final int numWorkers = Math.min(8, Runtime.getRuntime().availableProcessors());

        double[] A = randomMatrix(N, M, 42L);
        double[] B = randomMatrix(M, P, 99L);
        double[] C = new double[N * P];
        double[] rowNorms = new double[N];

        ExecutorService executor = Executors.newFixedThreadPool(numWorkers);

        CyclicBarrier barrierAfterMultiply = new CyclicBarrier(numWorkers);
        CyclicBarrier barrierAfterNorms = new CyclicBarrier(numWorkers);

        int rowsPerWorker = (N + numWorkers - 1) / numWorkers;
        List<Future<?>> futures = new ArrayList<>();

        for (int w = 0; w < numWorkers; w++) {
            final int startRow = w * rowsPerWorker;
            final int endRow = Math.min(N, startRow + rowsPerWorker);
            if (startRow >= endRow) break;

            futures.add(executor.submit(new Worker(A, B, C, rowNorms,
                    startRow, endRow, N, M, P, barrierAfterMultiply, barrierAfterNorms)));
        }

        for (Future<?> f : futures) f.get(); // surface exceptions
        executor.shutdown();
        if (!executor.awaitTermination(2, TimeUnit.MINUTES))
            executor.shutdownNow();

        System.out.println("Parallel multiply+norms complete.");
    }

    static class Worker implements Runnable {
        private final double[] A, B, C, norms;
        private final int startRow, endRow, N, M, P;
        private final CyclicBarrier barrier1, barrier2;

        Worker(double[] A, double[] B, double[] C, double[] norms,
```

```java
               int startRow, int endRow, int N, int M, int P,
               CyclicBarrier barrier1, CyclicBarrier barrier2) {
            this.A = A; this.B = B; this.C = C; this.norms = norms;
            this.startRow = startRow; this.endRow = endRow;
            this.N = N; this.M = M; this.P = P;
            this.barrier1 = barrier1; this.barrier2 = barrier2;
        }

        @Override
        public void run() {
            try {
                // Phase 1: compute assigned rows of C
                for (int i = startRow; i < endRow; i++) {
                    int rowA = i * M;
                    int rowC = i * P;
                    for (int k = 0; k < M; k++) {
                        double a = A[rowA + k];
                        int rowB = k * P;
                        for (int j = 0; j < P; j++) {
                            C[rowC + j] += a * B[rowB + j];
                        }
                    }
                }
                barrier1.await(); // memory fence + sync

                // Phase 2: compute row norms
                for (int i = startRow; i < endRow; i++) {
                    int rowC = i * P;
                    double s = 0.0;
                    for (int j = 0; j < P; j++) {
                        double v = C[rowC + j];
                        s += v * v;
                    }
                    norms[i] = Math.sqrt(s);
                }
                barrier2.await();

            } catch (InterruptedException ie) {
                Thread.currentThread().interrupt();
            } catch (BrokenBarrierException bbe) {
                System.err.println("Barrier broken in worker for rows "
                    + startRow + "-" + (endRow - 1));
            }
        }
    }

    static double[] randomMatrix(int rows, int cols, long seed) {
        Random rnd = new Random(seed);
        double[] m = new double[rows * cols];
        for (int i = 0; i < m.length; i++) m[i] = rnd.nextDouble();
        return m;
    }
}
```

Listing 1: Parallel matrix multiplication with CyclicBarrier

4.1 Key design notes

- **Flat arrays**: one-dimensional ‘double[]‘ with row-major indexing ‘i*cols + j‘.
- **Row-block partitioning**: lower synchronization and simple ownership semantics.
- **Barriers**: `barrier.await()` acts as a synchronization point and memory fence.

- **Diagnostics**: stored separately to avoid clobbering results.

5 Alternatives: invokeAll, Phaser, ForkJoin sketches

5.1 invokeAll + Callable

Each task returns its computed chunk (no shared writes during computation). Main thread merges results.

5.2 Phaser

Use ‘Phaser‘ for dynamic registration and many iterative phases. API: ‘phaser.arriveAndAwaitAdvance()‘.

5.3 ForkJoin recursive tiling

Implement ‘RecursiveAction‘ that splits blocks until base-case, then multiplies. Leverages work-stealing for load balance.

6 Verification, numeric tolerances, and correctness

6.1 Verification

Compare parallel result to a sequential reference for small matrices. Use checksums for quick detection.

6.2 Numeric tolerances

Use absolute and relative tolerances:

$$|x - y| \leq \mathrm{atol} + \mathrm{rtol} \cdot |y|.$$

Typical values depend on application; start with ‘atol=1e-12‘, ‘rtol=1e-9‘ for double.

7 Benchmarking and profiling

7.1 Warm-up

Run several warm-up iterations to allow JIT optimizations.

7.2 Measurement

Use ‘System.nanoTime()‘ for rough timing; use JMH for rigorous microbenchmarks. Collect wall-clock, CPU, memory, and cache metrics.

8 Memory layout, cache and false sharing

8.1 Row-major indexing

Store matrices as ‘double[]‘ and index by ‘i*cols + j‘ for contiguous row access.

8.2 False sharing mitigation

Assign coarser-grain chunks, add padding to per-thread buffers, avoid threads writing adjacent small regions.

8.3 Tile sizing

Rough heuristic:

$$b \approx \sqrt{\frac{\text{cacheBytes}}{3 \times 8}}.$$

Tune experimentally.

9 Error handling and production patterns

- Surface worker exceptions via ‘Future.get()‘.
- If barrier breaks, abort centrally with ‘executor.shutdownNow()‘.
- Use timed ‘await(timeout, unit)‘ if stalls are possible.

10 Advanced notes (brief)

Cache-oblivious recursion offers multi-level cache friendliness without explicit tile tuning, but tuned BLAS kernels still outperform general recursive code on most CPUs.

11 Production deployment recommendations

For critical workloads:

- Prefer vendor BLAS (MKL/OpenBLAS) or GPU libraries (cuBLAS).
- Profile to find whether compute, memory bandwidth, or synchronization is the bottleneck.
- Keep concurrency coarse-grained; use proven libraries for heavy lifting.

12 References & resources

- V. Strassen, “Gaussian elimination is not optimal”, *Numerische Mathematik*, 1969.
- BLAS: `http://www.netlib.org/blas/`
- Intel MKL: `https://software.intel.com/onemkl`
- NVIDIA cuBLAS: `https://developer.nvidia.com/cublas`
- JMH: Java Microbenchmark Harness documentation.

13 Checklist: prototype → production

1. Validate correctness on small matrices.
2. Warm-up and benchmark (JMH for accuracy).
3. Profile (CPU, memory, cache).
4. Tune tile size and loop order.
5. Check for false sharing and widen granularity if needed.
6. Consider native BLAS for production.