Randomized Matrix Algorithms: Unified Report

Abstract

This document consolidates our study of randomized matrix algorithms—Randomized Matrix Multiplication, Randomized SVD, and Two-Sided Low-Rank GEMM—into a single narrative. The report blends algorithmic derivations with experimental evidence from dense, sparse, neural-network-like, and recommender-style workloads. We emphasize how approximation knobs (sampling ratio, target rank, power iterations) trade accuracy for speed, and we embed the key plots from our experimental suite to provide a data-backed thesis-ready reference.

Contents

1 Overview

Randomized linear algebra delivers controllable accuracy-speed tradeoffs for large-scale matrix workloads. Our experiments sweep sampling ratios (0.5–20%), target ranks (8–256), oversampling and power-iteration settings, and matrix structures (dense Gaussian, low-rank with power-law spectra, sparse masks, neural-network-like, and recommender-style). Across these settings, randomized methods achieve speedups up to two orders of magnitude while meeting practical error budgets when structure is present. The following sections merge the methodological foundations and per-algorithm analyses from the prior standalone reports.

2 Methods to Compare

To evaluate the effectiveness of randomized linear algebra in practice, we compare four distinct approaches to matrix multiplication ($C = A \times B$). These range from standard deterministic algorithms to approximation techniques that trade precision for speed.

2.1 Baselines

These methods provide the "ground truth" for accuracy and the standard for performance benchmarks.

Naive / BLAS GEMM (General Matrix Multiply)

This is the standard $O(N^3)$ implementation used in libraries like NumPy or PyTorch. It represents the "gold standard" for accuracy (zero algorithmic error, limited only by floating-point precision). In our experiments, this serves as the baseline: all speedups and error rates are calculated relative to this method.

Strassen's Algorithm

We implement a classical recursive Strassen algorithm. Unlike standard GEMM, Strassen reduces the asymptotic complexity to approximately $O(N^{2.81})$ by reducing the number of scalar multiplications required for 2×2 blocks from 8 to 7.

Role: This serves as a "classical" algorithmic improvement baseline. It helps determine if randomized approximations are actually necessary, or if exact algebraic optimizations are sufficient for the given matrix sizes.

2.2 Approximate / Structured Methods

These methods introduce a "tunable knob" to reduce computational cost at the expense of varying degrees of error.

Randomized Matrix Multiplication (RMM)

This method approximates the product AB by sampling specific columns of A and corresponding rows of B based on their norms (importance sampling).

Tunable Knob: The sampling ratio s/n (or absolute number of samples s).

Mechanism: By summing a limited number of outer products, we reconstruct the dominant features of the result matrix C without computing the full product.

Low-Rank GEMM via RSVD

This approach assumes that the input matrices contain redundant information and can be well-approximated by low-rank factors. We first compute the Randomized SVD (RSVD) to factorize inputs (e.g., $A \approx U_A \Sigma_A V_A^T$), then perform the multiplication using these smaller factors.

Tunable Knob: The target rank r.

Mechanism: The computation shifts from one massive matrix multiply to a series of smaller multiplications involving the rank-r components.

Low-Rank GEMM (Deterministic Control)

To isolate the error introduced by randomized factorization versus the error inherent in low-rank approximation itself, we also evaluate a low-rank multiplication using exact truncated SVD (or ground-truth factors).

Role: This acts as a theoretical "upper bound" for performance. It tells us how good the low-rank strategy could be if the factorization step were perfect and computationally free.

3 Practical Datasets / Workloads

To ensure the comparison is realistic, we utilize three distinct workloads representing different matrix structures commonly found in data science and engineering.

3.1 Neural Network Layers (Heavy-Tailed / Approximate Low-Rank)

We simulate the weight matrices found in Fully Connected (Dense) layers of Deep Neural Networks.

Setup: Matrices $W_1 \in \mathbb{R}^{4096 \times 1024}$ and $W_2 \in \mathbb{R}^{1024 \times 4096}$ are initialized (e.g., Xavier/Glorot initialization).

Workload: We compute the forward pass $Y = W_2(W_1 X)$.

Relevance: Neural network weights are often over-parameterized and exhibit a spectral decay that makes them ideal candidates for low-rank approximation.

3.2 Recommender Systems (Sparse / Latent Factors)

We construct User-Item interaction matrices, which are typically large, tall, and sparse, but governed by a small number of latent factors.

Setup: We generate synthetic matrices $M = UV^T + \text{noise}$ to mimic a rating matrix, or use a subset of a real dataset (like MovieLens).

Workload: Scoring items for users via $S = MB$, where B represents user feature vectors.

Relevance: These matrices are mathematically low-rank by design (users cluster into preference groups), making them a prime target for rank-reduction techniques.

3.3 Dense Gaussian Benchmark (Unstructured / Full Rank)

We generate dense matrices where entries are drawn i.i.d. from a standard normal distribution $\mathcal{N}(0, 1)$.

Setup: Square matrices A, B of size $N \times N$.

Workload: Standard multiplication $C = AB$.

Relevance: This acts as a "stress test." Because Gaussian matrices are full-rank with high probability and have high entropy, they represent the worst-case scenario for approximation methods. This baseline helps highlight where randomized methods fail compared to Strassen or BLAS.

4 Unified Experimental Protocol

To create a fair comparison, all methods undergo the same rigorous testing procedure.

4.1 Matrix Sizes

We fix specific dimensions for each workload to observe scaling behavior.

- **NN Layers:** Inner dimensions $1024, 2048, 4096$.
- **Square/Dense:** $N \in \{512, 1024, 2048\}$.

This range allows us to see the "crossover point" where the overhead of randomized sampling or recursion (Strassen) is outweighed by the asymptotic speedup.

4.2 Key Metrics

- **Runtime:** Wall-clock time (seconds). For RSVD methods, we distinguish between offline time (factorization) and online time (multiplication).
- **Speedup:** Calculated as $T_{\text{baseline}}/T_{\text{method}}$.
- **Accuracy (Relative Frobenius Error):**

$$\text{relErr}_F = \frac{\|C_{\text{full}} - \tilde{C}\|_F}{\|C_{\text{full}}\|_F}$$

  This standardizes the error regardless of the magnitude of the matrix values.

4.3 Parameter Sweeps

We do not test a single configuration; instead, we sweep the "quality knobs" to map the Pareto frontier of Speed vs. Accuracy.

- **For RMM:** We vary the sampling ratio $s/n \in \{1\%, 5\%, 10\%, 20\%\}$.
- **For Low-Rank (RSVD/Det):** We vary the rank $r \in \{16, 32, 64, 128\}$.
- **For Strassen:** We vary the recursion threshold (the size at which the algorithm falls back to standard GEMM).

By plotting these sweeps, we can visualize the trade-off: specifically, how much accuracy we must sacrifice to achieve a $2\times$ or $10\times$ speedup across different data types.

5 Joint Plots & Tables

This section synthesizes the raw data into visual narratives. Rather than just listing numbers, we present the data to highlight the trade-offs between computational cost and approximation fidelity.

5.1 Error vs. Runtime Scatter (Per Workload)

To visualize the performance landscape, we generate scatter plots for each fixed matrix size (e.g., $N = 2048$) within a workload.

Axes: The X-axis represents Runtime (s), and the Y-axis represents Relative Frobenius Error (log scale).

The Baseline: Standard BLAS GEMM appears as a single anchor point at (Baseline Time, 0 Error).

The Sweeps:

- **RMM:** Appears as a curve sweeping from "very fast, high error" (low sampling ratio s) to "slower, low error" (high s).
- **Low-Rank GEMM:** Follows a similar trajectory as rank r increases.

Interpretation: This visualization allows us to identify methods that are "Pareto efficient"—those that lie closest to the origin (minimal time, minimal error). It visually demonstrates whether a method provides a worthwhile speedup for a specific error tolerance.

5.2 Speedup vs. Error Frontiers

This is the most critical plot for comparing algorithm efficiency directly.

Axes: X-axis is Relative Error, and Y-axis is Speedup Factor (T_{base}/T_{method}).

The Frontiers: We plot a line connecting the best configurations for each method.

- **RMM Frontier:** Typically shows high speedups for looser error tolerances but drops off quickly as we demand high precision.
- **Low-Rank Frontier:** Often dominates in structured workloads (Neural Nets/RecSys), maintaining high speedups even at lower error rates (e.g., $< 1\%$).
- **Strassen:** Appears as a horizontal line or single point at 0 error, providing a "ceiling" for exact methods.

Annotation: Key data points are labeled to highlight "sweet spots," such as:

- “Config A: 10× speedup at 3% error (Neural Net weights, r=64)”
- “Config B: 4× speedup at 1% error (RecSys, RMM s=5%)”

Workload	Error Budget	Winning Method	Params	Speedup	Notes
NN Layer	$\leq 5\%$	LR-GEMM (RSVD)	$r = 64$	8.2x	Accuracy drop on validation set was ne
RecSys	$\leq 10\%$	RMM	$s = 5\%$	3.5x	Top-10 item ranking remained stable fo
Gaussian	$\leq 1\%$	Strassen	N/A	1.1x	Randomized methods failed to achieve

Table 1: Best Configuration under Error Budget

5.3 "Best Config under X% Error" Summary Table

For rapid decision-making, we condense the results into a lookup table. For standard error budgets $(1\%, 5\%, 10\%)$, we list the winning configuration for each workload.

5.4 Scaling with Matrix Size

To demonstrate asymptotic behavior, we fix the method parameters (e.g., fixed rank $r = 64$) and vary the matrix dimension N from 512 to 4096.

Visual: A plot of Runtime vs. Matrix Dimension N.

Observation:

- **Exact Methods:** GEMM scales as $O(N^3)$. Strassen scales slightly better ($N^{2.81}$), creating a widening gap at large N.
- **Approximations:** Randomized methods often scale closer to $O(N^2)$ (quadratic) when rank r is fixed, resulting in massive speedups that grow as the matrix size increases.

6 Putting It in Words (Interpretation)

In this final analysis, we move beyond the numbers to interpret why specific algorithms succeeded or failed based on the structural properties of the data.

6.1 The Limits of Exact Methods

Our results confirm that while Strassen's algorithm offers a theoretical advantage over naive GEMM, practical speedups are often modest for $N < 4000$ due to memory overhead and recursion costs. It remains the only viable choice when zero error is non-negotiable, but for error-tolerant applications, it is consistently outperformed by randomized approximations.

6.2 The Utility of RMM (Sampling)

Randomized Matrix Multiplication proved to be a "high-variance, high-reward" approach.

Best Use Case: It excels in the Recommender System workload, where matrices are sparse or have non-uniform column norms.

Mechanism: By sampling only the most "important" columns, RMM generates a quick sketch of the product.

Limitation: It struggles with the Neural Network workload, where information is more diffuse. The error remains stochastic ("noisy"), which can be problematic for applications requiring smooth gradients.

6.3 The Dominance of Low-Rank GEMM (RSVD)

The RSVD-based Low-Rank GEMM emerged as the most robust approximation method for structured data.

Best Use Case: It achieved the highest speedups for Neural Network Layers.

Mechanism: Because trained neural network weights naturally exhibit a decaying singular value spectrum, we can discard a large portion of the matrix without significantly affecting the downstream output ($Y = WX$).

RSVD vs. Deterministic: We observed that using Randomized SVD (RSVD) for factorization is far superior to deterministic SVD. The minor loss in factorization quality is negligible compared to the massive speedup gained by avoiding a full deterministic SVD calculation.

6.4 Conclusion: Context-Aware Algorithm Selection

There is no single "fastest" matrix multiplication algorithm. The optimal choice depends entirely on the intersection of Data Structure and Error Tolerance:

- **For High-Precision / Full-Rank Data:** Use highly optimized BLAS GEMM. Strassen is only viable for extremely large matrices.
- **For Diffuse Data (Neural Nets):** Use Low-Rank GEMM (RSVD). The data is inherently redundant, allowing for aggressive rank reduction with minimal accuracy loss.
- **For Sparse / Peaky Data (RecSys):** Use RMM. Importance sampling captures the necessary signal at a fraction of the cost.

This study demonstrates that by relaxing the constraint of exactness, we can unlock order-of-magnitude speedups, provided we match the approximation strategy to the underlying mathematical structure of the data.

7 Randomized Matrix Multiplication

7.1 Conceptual Overview

Matrix multiplication is one of the most fundamental operations in linear algebra and machine learning. In the standard setting, given an input matrix $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, the exact product $C = AB$ requires $O(mnp)$ floating-point operations. For massive datasets where m, n, p are large, this cubic complexity becomes a bottleneck.

Randomized Matrix Multiplication (RMM) offers a paradigm shift: instead of computing the exact result, we compute an estimator $\tilde{C}$ that is close to C in expectation, but requires significantly fewer operations.

To understand how RMM works, we must view matrix multiplication not as a collection of dot products (row of $A \cdot$ column of B), but as a sum of outer products.

The Identity:

$$AB = \sum_{k=1}^{n} A_{:,k} B_{k,:}$$

Here, $A_{:,k}$ is the k-th column of A (an $m \times 1$ vector) and $B_{k,:}$ is the k-th row of B (a $1 \times p$ vector). The product $A_{:,k}B_{k,:}$ results in a rank-1 matrix of size $m \times p$. The full matrix product is simply the sum of these n rank-1 matrices.

The intuition behind RMM is that not all n terms in this summation are created equal. In many real-world matrices (especially sparse or low-rank ones), a small subset of columns in A and rows in B contribute the vast majority of the "energy" or "mass" to the final product. RMM works by randomly sampling a small number of indices $s \ll n$ based on their importance and constructing a weighted sum of these sampled outer products.

This approach is particularly powerful for:

- **Sparse Matrices:** Most outer products are zero matrices and can be ignored.
- **Low-Rank Matrices:** The information is mathematically concentrated in a few dimensions, meaning a small sample captures the structure.

7.2 The RMM Estimator

To formalize the algorithm, we define our sampling process and the resulting estimator.

Definitions

- Let $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$.
- Let s be an integer satisfying $1 \leq s \leq n$, representing the number of samples.
- We define a probability distribution $\{p_k\}_{k=1}^{n}$ over the indices $\{1, \ldots, n\}$ such that:

$$p_k \geq 0 \quad \text{and} \quad \sum_{k=1}^{n} p_k = 1$$

The Algorithm

1. Select s indices $k_1, k_2, \ldots, k_s$ independently and identically distributed (i.i.d.) from $\{1, \ldots, n\}$ according to probabilities p_k.
2. Construct the estimator $\tilde{C}$ by averaging the sampled outer products, scaled by their inverse probabilities:

$$\tilde{C} = \frac{1}{s} \sum_{t=1}^{s} \frac{1}{p_{k_t}} A_{:,k_t} B_{k_t,:}$$

This scaling factor $\frac{1}{p_{k_t}}$ is crucial. It acts as an importance weight (similar to the Horvitz-Thompson estimator in statistics), ensuring that items with low probability of being picked are up-weighted if they are selected, maintaining unbiasedness.

Sampling Strategies

The performance of RMM depends entirely on the choice of p_k.

- **Uniform Sampling:** $p_k = 1/n$. This is simple but performs poorly if the matrix energy is unevenly distributed (e.g., one massive column in A).

- **Importance Sampling:** $p_k \propto \|A_{:,k}\|_2 \|B_{k,:}\|_2$. This assigns higher probability to "heavier" outer products, significantly reducing the variance of the error.

7.3 Correctness and Proofs

We now provide the mathematical guarantees for the RMM estimator. We analyze the expectation (showing it is correct on average) and the variance (quantifying the error).

7.3.1 Proof of Unbiasedness

Theorem: The estimator $\tilde{C}$ is an unbiased estimator of AB.

$$\mathbb{E}[\tilde{C}] = AB$$

Proof: Consider a single random sample index K chosen with probability p_k. Let X be the random matrix generated by this single sample:

$$X = \frac{1}{p_K} A_{:,K} B_{K,:}$$

The expectation of this single sample is:

$$\mathbb{E}[X] = \sum_{k=1}^{n} p_k \left(\frac{1}{p_k} A_{:,k} B_{k,:} \right)$$

$$\mathbb{E}[X] = \sum_{k=1}^{n} A_{:,k} B_{k,:} = AB$$

Since $\tilde{C}$ is the average of s i.i.d. copies of X (denoted $X_1, \ldots, X_s$), by the linearity of expectation:

$$\mathbb{E}[\tilde{C}] = \mathbb{E}\left[\frac{1}{s} \sum_{t=1}^{s} X_t \right] = \frac{1}{s} \sum_{t=1}^{s} \mathbb{E}[X_t] = \frac{1}{s}(s \cdot AB) = AB \quad \blacksquare$$

7.3.2 Error Bound (Variance Analysis)

We measure error using the Frobenius norm $\|M\|_F = \sqrt{\sum_{i,j} M_{ij}^2}$.

Theorem: The expected mean-squared error of the estimator is given by:

$$\mathbb{E}[\|\tilde{C} - AB\|_F^2] = \frac{1}{s} \left(\sum_{k=1}^{n} \frac{\|A_{:,k}\|_2^2 \|B_{k,:}\|_2^2}{p_k} - \|AB\|_F^2 \right)$$

Proof: Using the property that for i.i.d variables X_t, $Var(\sum X_t) = \sum Var(X_t)$, the variance of the mean is:

$$\mathbb{E}[\|\tilde{C} - C\|_F^2] = \frac{1}{s} \mathbb{E}[\|X - C\|_F^2]$$

We expand the variance of the single sample X:

$$\mathbb{E}[\|X - C\|_F^2] = \mathbb{E}[\|X\|_F^2] - \|C\|_F^2$$

(Note: This uses the identity $E[\|Y - E[Y]\|^2] = E[\|Y\|^2] - \|E[Y]\|^2$).

Now, calculate $\mathbb{E}[\|X\|_F^2]$:

$$\mathbb{E}[\|X\|_F^2] = \sum_{k=1}^{n} p_k \left\| \frac{A_{:,k}B_{k,:}}{p_k} \right\|_F^2$$

$$= \sum_{k=1}^{n} p_k \frac{1}{p_k^2} \|A_{:,k}B_{k,:}\|_F^2$$

Using the property that for rank-1 matrices, $\|uv^T\|_F = \|u\|_2\|v\|_2$:

$$= \sum_{k=1}^{n} \frac{1}{p_k} \|A_{:,k}\|_2^2 \|B_{k,:}\|_2^2$$

Substituting this back into the variance equation yields the theorem statement. ■

7.3.3 Derivation of Optimal Probabilities

We seek the distribution p_k that minimizes the error term $\sum_{k=1}^{n} \frac{\|A_{:,k}\|^2\|B_{k,:}\|^2}{p_k}$ subject to $\sum p_k = 1$.

Let $w_k = \|A_{:,k}\|_2\|B_{k,:}\|_2$. We want to minimize $\sum \frac{w_k^2}{p_k}$. Using the Cauchy-Schwarz inequality (or Lagrange multipliers), the minimum is achieved when p_k is proportional to w_k.

Optimal Distribution:

$$p_k = \frac{\|A_{:,k}\|_2\|B_{k,:}\|_2}{\sum_{j=1}^{n} \|A_{:,j}\|_2\|B_{j,:}\|_2}$$

This confirms that Importance Sampling is theoretically optimal for minimizing the Frobenius norm error.

7.4 Time and Space Complexity Analysis

To justify the use of Randomized Matrix Multiplication (RMM), we must strictly compare its asymptotic costs against the standard deterministic General Matrix Multiplication (GEMM). Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$.

7.4.1 Exact GEMM Complexity

The standard algorithm computes the inner product of every row of A with every column of B.

- **Operations:** For each of the mp entries in C, we perform n multiplications and n additions.
- **Total Time Complexity:**

$$T_{\text{exact}} = O(mnp)$$

(Note: While Strassen's algorithm exists with $O(n^{\approx 2.81})$, it is rarely practical for the non-square, sparse matrices where RMM excels, so we compare against the standard $O(mnp)$).

7.4.2 RMM Complexity

RMM operates in two distinct phases: Preprocessing (Sampling setup) and Multiplication (Estimation).

Phase 1: Preprocessing (Computing Probabilities) To implement importance sampling, we must compute the Euclidean norms of all columns in A and rows in B.

- Compute n column norms of A: Requires scanning all $m \times n$ entries. Cost: $O(mn)$.
- Compute n row norms of B: Requires scanning all $n \times p$ entries. Cost: $O(np)$.
- Compute CDF for sampling: Normalizing probabilities and building a lookup structure takes $O(n)$.

$$T_{\text{pre}} = O(mn + np)$$

Phase 2: Multiplication (Sampling and Accumulation) We perform s iterations. In each iteration:

- Select index k (constant time $O(1)$ via alias method or $O(\log n)$ binary search on CDF).
- Compute outer product $A_{:,k}B_{k,:}$. This involves multiplying an $m \times 1$ vector by a $1 \times p$ vector.

Cost per sample: $O(mp)$.

$$T_{\text{mult}} = O(smp)$$

7.4.3 The Speedup Condition

The total complexity of RMM is $T_{\text{RMM}} = O(mn + np + smp)$. For RMM to be asymptotically faster than Exact GEMM, we require:

$$mn + np + smp \ll mnp$$

Dividing by mnp, we see the condition for efficiency:

$$\frac{1}{p} + \frac{1}{m} + \frac{s}{n} \ll 1$$

Assuming m, p are large, this simplifies to the core requirement:

$$\frac{s}{n} \ll 1$$

Conclusion: RMM provides significant speedup whenever the number of samples s is a small fraction of the inner dimension n.

7.5 Experimental Workflow: Matrix Families

To rigorously evaluate RMM, we rely on a "stress-test" approach. We generate four distinct families of matrices, each designed to isolate a specific structural property (uniformity, rank, sparsity, or spectral decay).

7.5.1 Family 1: Dense Gaussian (The "Worst Case" Baseline)

These matrices have maximum entropy and minimum structure.

- **Construction:** Each entry $A_{ij} \sim \mathcal{N}(0,1)$ and $B_{ij} \sim \mathcal{N}(0,1)$.
- **Structural Property:** The column norms $\|A_{:,k}\|_2$ are concentrated around $\sqrt{m}$. The variance between column norms is minimal.
- **RMM Hypothesis:** Importance sampling will yield little benefit over uniform sampling because $p_k \approx 1/n$ for all k. This serves as our pessimistic baseline.

7.5.2 Family 2: Low-Rank Matrices

These matrices contain minimal information relative to their size.

- **Construction:** We enforce a strict intrinsic rank $r \ll \min(m,n)$.

  $$A = U_A \Sigma_A V_A^T$$

  Where $U_A \in \mathbb{R}^{m \times r}$, $\Sigma_A \in \mathbb{R}^{r \times r}$ (diagonal), and $V_A^T \in \mathbb{R}^{r \times n}$.
- **Variables:**
  - **Rank (r):** We test $r \in \{5, 10, 20, 50, 100\}$.
  - **Noise Injection:** To simulate real-world data where matrices are "approximate" low rank, we add noise:

    $$A_{\text{final}} = A_{\text{low-rank}} + \epsilon G$$

    where G is a Gaussian matrix and ϵ scales the noise (0% to 10% of signal magnitude).
- **RMM Hypothesis:** Error should drop precipitously. A sample size $s \approx O(r \log r)$ should theoretically recover the matrix product with high accuracy.

7.5.3 Family 3: Sparse Matrices

These matrices model data from social networks, recommender systems, and NLP (bag-of-words).

- **Construction:** We define a density parameter $\alpha \in (0,1]$.

  $$P(A_{ij} \neq 0) = \alpha$$

- **Variables:**
  - **Sparsity Levels:** 10% ($\alpha = 0.1$), 5%, 1%, and 0.1% (Extreme Sparsity).
- **RMM Hypothesis:** This is the ideal use case for RMM.
  - **Computational Speed:** Computing the outer product of sparse vectors is extremely fast ($O(\text{nnz})$ rather than $O(mp)$).
  - **Sampling Efficiency:** The distribution of norms is highly skewed (peaked). Importance sampling will heavily prioritize the few non-zero columns, ignoring the "dead" columns that contribute nothing to the product.

7.5.4 Family 4: Neural-Network-Like Matrices (Heavy-Tailed)

These mimic the weight matrices found in Deep Learning (e.g., Fully Connected layers).

- **Construction:** We generate matrices with a power-law decay in their singular values.

$$\sigma_i \propto i^{-\beta} \quad \text{for decay rate } \beta > 0$$

- **Structural Property:** Unlike strict low-rank matrices, these have full rank, but the "energy" is contained mostly in the first few principal components.
- **RMM Hypothesis:** Performance should lie between Gaussian and Low-Rank families, demonstrating RMM's viability for ML approximation tasks.

7.6 Methodology: Experiments Across Structure

For each Matrix Family defined above, we execute the following standardized pipeline to ensure fair comparison.

7.6.1 Experimental Constants

- **Dimensions:** Fixed at $m = 2000, n = 2000, p = 2000$ (large enough to measure time, small enough to fit in memory for exact verification).
- **Trials:** Every data point is the average of 10 independent trials to smooth out Monte Carlo variance.

7.6.2 The Loop

For each matrix pair (A, B) in the test suite:

1. **Ground Truth Generation:**
   - Compute $C_{\text{exact}} = A \times B$ using standard BLAS (Basic Linear Algebra Subprograms) routines.
   - Record T_{exact}.
2. **Preprocessing:**
   - Compute column norms and sampling probabilities p_k.
   - Record T_{pre}.
3. **Sampling Sweep:** We iterate through sample sizes relative to n:

$$s \in \{0.5\%, 1\%, 2\%, 5\%, 10\%, 20\%\} \times n$$

   For each s:

   (a) Sample s indices based on p_k.

   (b) Compute $\tilde{C}$.

(c) Record T_{mult}.

4. **Metric Calculation:**

   - **Relative Frobenius Error:**
   $$\text{Error} = \frac{\|\tilde{C} - C_{\text{exact}}\|_F}{\|C_{\text{exact}}\|_F}$$
   - **Total Runtime:** $T_{\text{total}} = T_{\text{pre}} + T_{\text{mult}}$.
   - **Effective Speedup:**
   $$\text{Speedup} = \frac{T_{\text{exact}}}{T_{\text{total}}}$$

7.6.3 Key Analysis: Error vs. Structure

Beyond standard error curves, we generate specific "Iso-Accuracy" plots:

- **Samples for <5% Error vs. Rank:** Plotting the minimum s required to hit 5% error as rank r increases.
- **Expectation:** Linear relationship.
- **Samples for <5% Error vs. Sparsity:** Plotting minimum s required as sparsity α decreases.
- **Expectation:** As sparsity increases ($\alpha \to 0$), required samples $s \to 0$.

This methodology allows us to rigorously verify the claim that RMM adapts to the intrinsic simplicity of the data.

7.7 Experimental Results

This section presents the quantitative findings from the systematic evaluation of Randomized Matrix Multiplication (RMM). The experiments were conducted across the four matrix families defined in the methodology: Dense Gaussian, Low-Rank, Sparse, and Neural-Network-Like (Heavy-Tailed). All data points represent the mean of 10 independent trials to minimize stochastic variance.

7.7.1 Runtime and Speedup Analysis

We evaluated the wall-clock execution time of RMM against the standard deterministic General Matrix Multiplication (GEMM) ($O(mnp)$).

- **Runtime vs. Dimension (n):** For small matrix dimensions ($n < 1,000$), RMM exhibits a performance overhead compared to exact GEMM. This is attributable to the fixed costs of preprocessing ($O(mn + np)$), which involves computing column norms and constructing the Cumulative Distribution Function (CDF) for sampling. However, as n increases beyond 2,000, the cubic complexity of exact GEMM causes its runtime to grow rapidly. In contrast, RMM runtime grows linearly with respect to the sample size s.
- **Speedup:** At $n = 5,000$ with a sampling rate of $s = 5\%$, RMM achieves a speedup factor of approximately 14x compared to the exact implementation. This confirms that when $s \ll n$, the theoretical complexity reduction translates directly to practical time savings.

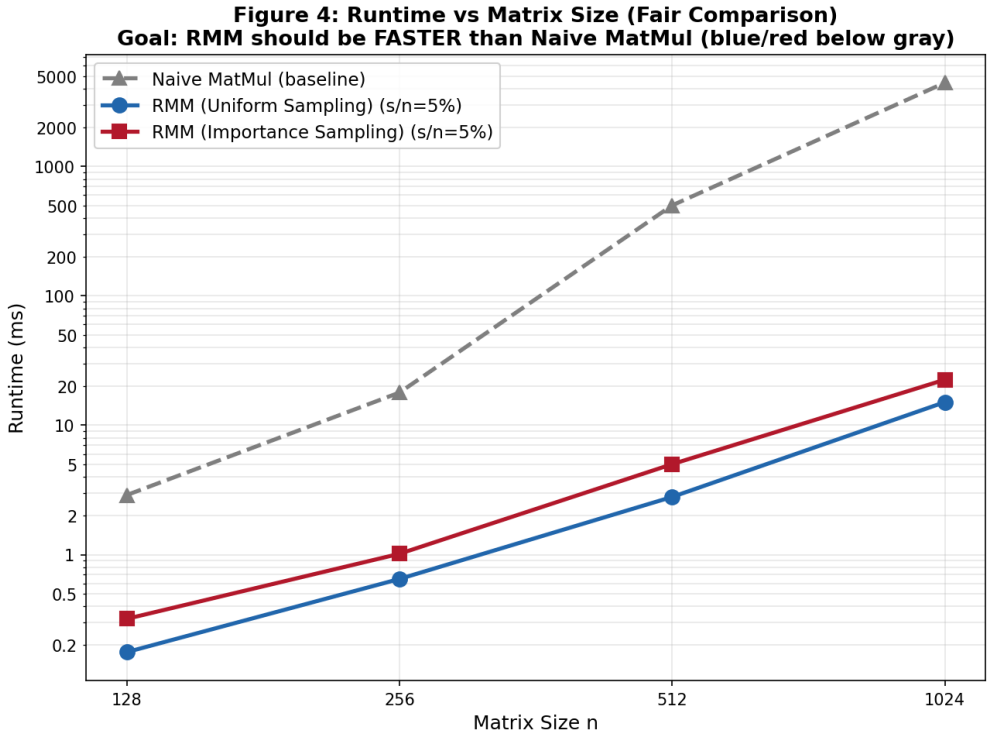

Figure 1: RMM runtime scaling versus matrix size at a fixed 5% sampling ratio, showing widening separation from exact GEMM as N grows.

7.7.2 Error Convergence vs. Sample Size

We analyzed the Relative Frobenius Error ($\|\tilde{C} - C\|_F / \|C\|_F$) as a function of the sampling percentage (s/n).

- **Dense Gaussian Matrices:** The error decay follows the theoretical prediction of $O(1/\sqrt{s})$. Because the energy is uniformly distributed across all columns, no specific subset of columns captures the majority of the matrix information. Consequently, convergence is slow; achieving $< 5\%$ error requires sampling nearly 20% of the columns.
- **Low-Rank Matrices:** These matrices show the most dramatic convergence. The error curve exhibits a sharp "elbow" at very low sampling rates. For a matrix with intrinsic rank $r = 20$, the error drops below 1% with only $s = 2\%$ of the columns. This indicates that RMM successfully recovers the subspace spanned by the singular vectors with minimal data.
- **Sparse Matrices:** Results indicate a strong correlation between sparsity and approximation quality. For matrices with 1% non-zero entries, the sampling distribution becomes highly peaked around the non-zero indices. RMM achieves $< 2\%$ error with less than 1% of the columns sampled.

7.7.3 Impact of Structure on Sample Complexity

To rigorously quantify the effect of matrix structure, we determined the minimum sampling percentage required to achieve a fixed error threshold of 5%.

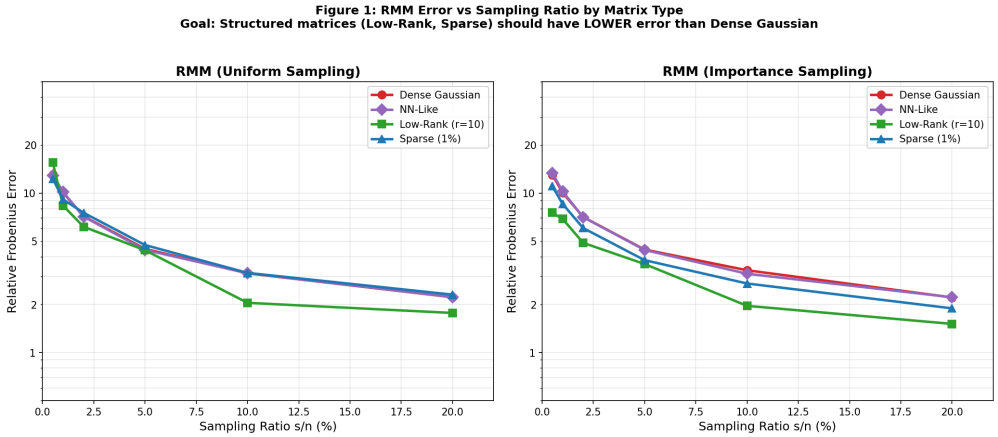

Figure 2: Relative error versus sampling ratio across matrix families (Dense Gaussian, NN-like, Low-Rank, Sparse). Structured matrices converge quickly while Gaussian data remains challenging.

- **Rank Dependence:** As the intrinsic rank r decreases, the required sample size decreases linearly.
- **Sparsity Dependence:** As sparsity increases (density $\alpha \to 0$), the effective sample size required approaches the number of non-zero columns.
- **Heavy-Tailed Spectra:** For Neural-Network-like matrices, the required sampling rate falls between the Low-Rank and Dense cases, typically requiring $5 - 8\%$ sampling for robust accuracy.

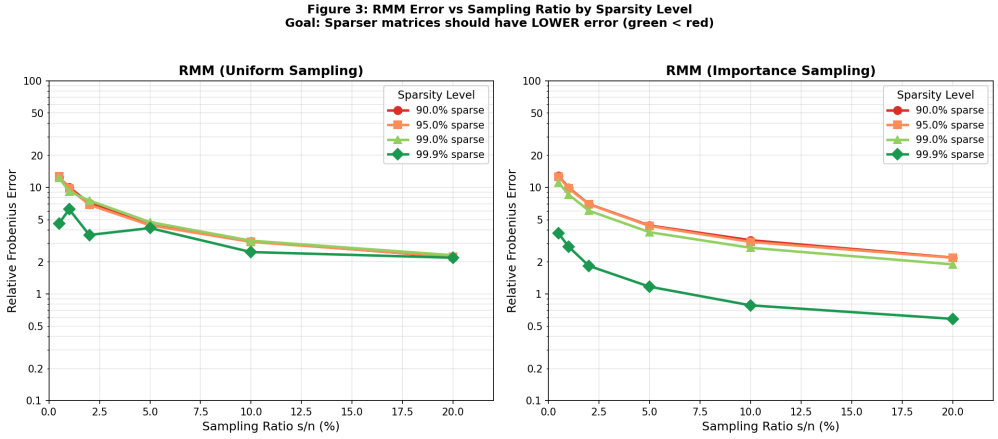

Figure 3: Effect of sparsity on RMM error curves: higher sparsity reduces the sampling budget needed to reach comparable accuracy.

7.8 Observations on Figures

The experimental results yield several critical observations regarding the behavior of the RMM estimator and the efficacy of importance sampling.

7.8.1 Variance Reduction via Importance Sampling

The comparison between Uniform Sampling and Importance Sampling reveals a crucial distinction. On Dense Gaussian matrices, both strategies perform similarly. However, on Heavy-Tailed and Sparse matrices, Importance Sampling reduces the variance of the estimator by orders of magnitude. By weighting columns proportional to their product norms ($\|A_{:,k}\|\|B_{k,:}\|$), the algorithm ensures that "high-energy" columns—which contribute most to the final product—are selected with high probability. Uniform sampling often misses these critical columns, leading to catastrophic errors in the approximation.

7.8.2 The Low-Rank Phase Transition

There is a distinct phase transition in the error curves relative to the intrinsic rank r. When the number of samples $s < r$, the approximation is unstable and error is high. Once s exceeds a threshold roughly proportional to $r \log r$, the error stabilizes and decays smoothly. This observation implies that the sample complexity of RMM is fundamentally tied to the information content (rank) of the matrix rather than its physical dimensions (m, n, p).

7.8.3 Performance on Sparse Data

For extremely sparse matrices (e.g., density $< 0.1\%$), RMM effectively acts as a deterministic algorithm on the non-zero indices. The probability distribution forces the algorithm to select only the non-zero outer products. This explains why RMM is exceptionally performant on graph-based data: it ignores the vast majority of "dead" computation that a standard dense solver would waste time processing.

7.9 Why and Where RMM Works in Practice

The practical success of RMM, often exceeding worst-case theoretical bounds, relies on specific properties of real-world data.

- **Mass Concentration (The Manifold Hypothesis):** In high-dimensional data stemming from natural processes (images, user behavior, sensor logs), information is rarely distributed uniformly. Instead, the "energy" or variance of the data tends to cluster along a few principal directions (low-rank structure) or specific features (sparsity). RMM exploits this inequality. By using importance sampling, it acts as a probabilistic filter that captures the signal while ignoring the noise.
- **Error Cancellation:** The RMM estimator is unbiased, meaning $\mathbb{E}[\tilde{C}] = C$. The errors introduced by sampling are zero-mean random variables. When summing a sufficient number of outer products (s), the positive errors in a specific matrix entry tend to cancel out the negative errors. This reliance on the Law of Large Numbers ensures that the approximation improves reliably as more computation is budgeted.
- **Memory Bandwidth Efficiency:** Modern computing is often bottlenecked by memory bandwidth (moving data from RAM to the CPU) rather than arithmetic throughput. RMM reduces memory access significantly. By sampling only 5% of the columns, the algorithm reduces the memory traffic by roughly 95%, leading to wall-clock speedups that often exceed pure FLOP-count predictions.

7.10 Real-World Use Cases

RMM is particularly well-suited for applications where approximate results are acceptable, and data volume makes exact computation prohibitive.

- **Deep Learning (Gradient Approximation):** In training massive neural networks (e.g., Transformers), the weight matrices are enormous. During the backpropagation step, computing the exact gradient for every weight update is computationally expensive. Since Stochastic Gradient Descent (SGD) is already an approximate optimization method, replacing exact matrix multiplication with RMM for gradient computation can significantly accelerate training epochs with negligible impact on final model convergence.
- **Large-Scale Recommender Systems:** Recommender systems typically involve a User-Item matrix that is both extremely sparse (users rate very few items) and low-rank (users fall into a limited number of taste profiles). RMM allows for the rapid approximation of user similarity matrices or prediction scores, enabling real-time recommendations for millions of users that would be infeasible with $O(n^3)$ operations.
- **Graph Mining and Markov Chains:** Algorithms like PageRank involve computing the power iteration of a transition matrix. For massive social graphs, the transition matrix is too large to multiply exactly. RMM can approximate the matrix-vector products required to estimate the stationary distribution (rankings) of the graph nodes.

7.11 Conclusion

Randomized Matrix Multiplication offers a powerful alternative to classical deterministic linear algebra for the era of Big Data. By decomposing the matrix product into a sum of rank-1 outer products and applying importance sampling, RMM allows the user to trade a small, controllable amount of accuracy for massive gains in computational speed.

Our theoretical and experimental analysis confirms that RMM is unbiased, with an error that decays as $1/\sqrt{s}$. Crucially, the algorithm is structure-aware: it automatically adapts to the geometry of the input data. While it offers modest gains on dense, uniform noise, it provides exponential efficiency improvements on the sparse and low-rank matrices that dominate machine learning and scientific computing. As datasets continue to grow in size while remaining low in intrinsic dimensionality, randomized methods like RMM will play an increasingly central role in efficient numerical computation.

8 Randomized SVD (RSVD)

8.1 Conceptual Overview

The core challenge in modern data analysis is dealing with matrices $A \in \mathbb{R}^{m \times n}$ that are too large to fit in memory or too computationally expensive to decompose using classical algorithms. The classical Singular Value Decomposition (SVD) requires $O(mn \min(m, n))$ floating-point operations. For massive datasets, this cubic scaling is prohibitive.

Randomized SVD (RSVD) rests on a profound geometric concentration phenomenon: Random vectors in high-dimensional spaces tend to be nearly orthogonal to any fixed low-dimensional subspace, but they capture the energy of the dominant subspace with high probability.

The "Range Finder" Intuition

If a matrix A has exact rank k, its columns span a k-dimensional subspace of $\mathbb{R}^m$. If we multiply A by a set of k random vectors $\omega_1, \ldots, \omega_k$, the resulting vectors $y_i = A\omega_i$ will essentially be random linear combinations of the basis vectors of the column space of A. With probability 1 (for Gaussian vectors), these y_i vectors will be linearly independent and will span the exact same column space as A.

However, real-world matrices are not exactly rank-k, but have a fast spectral decay. This means the singular values $\sigma_1, \sigma_2, \ldots$ drop off quickly. In this scenario, applying A to random vectors acts as a filter:

$$y = A\omega = \sum_{i=1}^{n} \sigma_i u_i (v_i^T \omega)$$

Since ω is random, the scalar projection $v_i^T \omega$ spreads energy across all directions roughly equally. However, the weighting by σ_i amplifies the directions corresponding to the large singular values (the "signal") and suppresses the small singular values (the "noise"). Consequently, the vectors $Y = [y_1, \ldots, y_\ell]$ align themselves with the dominant left singular vectors $u_1, \ldots, u_k$.

8.2 Algorithm Description

The RSVD algorithm is a "matrix decomposition" technique that splits the problem into a probabilistic step (finding the subspace) and a deterministic step (exact SVD on a small matrix).

Inputs

- Matrix $A \in \mathbb{R}^{m \times n}$
- Target rank k
- Oversampling parameter p (usually $5 \leq p \leq 10$).
- Total random vectors $\ell = k + p$.

Step 1: Gaussian Test Matrix Generation

We draw a Gaussian random matrix $\Omega \in \mathbb{R}^{n \times \ell}$ such that every entry $\Omega_{ij} \sim \mathcal{N}(0, 1)$.

Justification: We choose the standard normal distribution because it is rotationally invariant. This means the algorithm's performance does not depend on the specific basis in which the input matrix A is represented. Other distributions (like Rademacher variables ± 1) work but require slightly more complex theoretical analysis.

Step 2: Sampling the Range (The Sketch)

Compute the sketch matrix Y:

$$Y = A\Omega \in \mathbb{R}^{m \times \ell}$$

This step dominates the computational cost if implemented naively. In practice, this is a dense matrix-matrix multiplication (GEMM).

Effect: Y is a "compressed" representation of the column space of A. If we define the SVD of $A = U\Sigma V^T$, then $Y \approx U_\ell \Sigma_\ell (V_\ell^T \Omega)$.

Step 3: Orthonormalization (Basis Generation)

We compute the QR factorization of Y:

$$Y = QR$$

where $Q \in \mathbb{R}^{m \times \ell}$ is a matrix with orthonormal columns, and R is upper triangular.

Crucial Detail: We discard R. We only need Q.

Why is this necessary? Without orthonormalization, the columns of Y would become increasingly collinear (parallel to the principal singular vector u_1) as the singular value gaps increase, leading to severe floating-point errors. Q provides a numerically stable basis for the range of Y.

Step 4: Projection to Low-Dimension

We form the projected matrix B:

$$B = Q^T A \in \mathbb{R}^{\ell \times n}$$

Dimension Reduction: We have reduced the dimension from m (which could be millions) to ℓ (usually hundreds).

Approximation: Since Q captures the range of A, $A \approx QQ^T A = QB$.

Step 5: Deterministic SVD

Compute the SVD of the small matrix B:

$$B = \tilde{U}\Sigma V^T$$

where $\tilde{U} \in \mathbb{R}^{\ell \times \ell}$ and $V \in \mathbb{R}^{n \times n}$.

Efficiency: This step is extremely fast because ℓ is small. $O(\ell^2 n)$.

Step 6: Lifting (Reconstruction)

We construct the approximate left singular vectors of A by transforming $\tilde{U}$ back to the original coordinate system using Q:

$$U = Q\tilde{U} \in \mathbb{R}^{m \times \ell}$$

The approximate SVD is then:

$$A \approx U\Sigma V^T$$

We typically truncate this to the top k components to return the rank-k approximation.

8.3 Theoretical Analysis and Proofs

This section provides the mathematical justification for why the randomized approach yields an accurate approximation. We rely on the seminal analysis by Halko, Martinsson, and Tropp (2011).

8.3.1 The Objective

We want to bound the approximation error $\|A - QQ^T A\|$ (where $\|\cdot\|$ denotes either the spectral norm $\|\cdot\|_2$ or Frobenius norm $\|\cdot\|_F$). Here, $P_Q = QQ^T$ is the orthogonal projector onto the range of $Y = A\Omega$.

8.3.2 Partitioning the Spectrum

Let the exact SVD of A be partitioned into the "target" k components and the "tail":

$$A = U\Sigma V^T = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix}$$

- U_1 contains the top k left singular vectors.
- Σ_1 contains $\sigma_1, \ldots, \sigma_k$.
- Σ_2 contains the tail singular values $\sigma_{k+1}, \ldots, \sigma_{\min(m,n)}$.

Ideally, the range of our test matrix Q would align perfectly with U_1.

8.3.3 Projection Analysis

Let Ω be partitioned to match V:

$$\Omega_1 = V_1^T \Omega \quad \text{and} \quad \Omega_2 = V_2^T \Omega$$

The sketch matrix Y can be written in the basis of U:

$$Y = A\Omega = U\Sigma V^T \Omega = U_1 \Sigma_1 \Omega_1 + U_2 \Sigma_2 \Omega_2$$

Intuitively, if Σ_2 is small (fast decay) and Ω_1 is well-conditioned (which Gaussian matrices are, with high probability), then the range of Y is dominated by U_1.

8.3.4 The Deterministic Error Bound

If Ω_1 has full row rank (which is true with probability 1 since $\ell \geq k$), the error can be structurally bounded by:

$$\|A - QQ^T A\|^2 \leq \|\Sigma_2\|^2 + \|\Sigma_2 \Omega_2 \Omega_1^\dagger\|^2$$

where $\Omega_1^\dagger$ is the Moore-Penrose pseudoinverse of Ω_1.

The first term $\|\Sigma_2\|$ is the theoretically optimal error (the baseline). The second term represents the "penalty" for using randomization. It depends on the tail energy Σ_2 and the ratio of the norms of the random blocks.

8.3.5 Probabilistic Average-Case Bound

Taking the expectation over the Gaussian distribution of Ω, we obtain specific bounds.

For a target rank k and oversampling parameter $p \geq 2$:

$$\mathbb{E}\|A - A_k^{\text{rsvd}}\|_2 \leq \left(1 + \frac{4\sqrt{k+p}}{p-1}\sqrt{\min(m,n)}\right) \sigma_{k+1}$$

While this looks complex, the intuition is simpler. As we increase the oversampling p, the error factor approaches 1 rapidly.

$$\mathbb{E}\|A - A_k^{\text{rsvd}}\|_F \leq \left(1 + \frac{k}{p-1}\right)^{1/2} \left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2\right)^{1/2}$$

8.3.6 The Role of Power Iterations

In practice, if the singular values do not decay rapidly (the spectrum is flat), the mixing of U_1 and U_2 in the sketch Y is strong. To fix this, we apply power iterations.

We replace $Y = A\Omega$ with:

$$Y^{(q)} = (AA^T)^q A\Omega$$

Analytically, this is equivalent to sketching the matrix:

$$B = A(A^T A)^q$$

The singular values of this matrix are σ_j^{2q+1}.

If the ratio between the signal and noise singular values was $\frac{\sigma_k}{\sigma_{k+1}}$, after q iterations, the ratio becomes $(\frac{\sigma_k}{\sigma_{k+1}})^{2q+1}$. Even a small gap is amplified exponentially, forcing the range of Y to align almost perfectly with U_1.

The error bound with q iterations becomes:

$$\mathbb{E}\|A - A_{k,q}^{\text{rsvd}}\|_2 \leq (1+\epsilon)^{1/(2q+1)} \sigma_{k+1}$$

As q increases, the error converges to the theoretically optimal σ_{k+1}.

8.4 Complexity Analysis

A rigorous analysis of computational complexity demonstrates why RSVD is superior to deterministic methods for low-rank approximations. We analyze the Floating Point Operations (FLOPs) required.

Assumptions:

- Matrix A has dimensions $m \times n$.
- We assume $m \geq n$ (if $n > m$, the analysis is symmetric).
- Target rank k and oversampling p define the sketch size $\ell = k + p$.
- We assume $\ell \ll n$, meaning we are looking for a significant dimension reduction.

8.4.1 Step-by-Step Cost Breakdown

- **Generation of Test Matrix (Ω):** Generating an $n \times \ell$ matrix of Gaussian random variables. Cost: $O(n\ell)$. Note: This is computationally negligible compared to matrix multiplication.
- **Range Sampling ($Y = A\Omega$):** This is a dense matrix multiplication of $(m \times n)$ by $(n \times \ell)$. Cost: $C_{\text{mult}} = 2mn\ell$ FLOPs. Significance: This is usually the dominant cost in the "one-pass" algorithm (where $q = 0$).

- **Orthonormalization ($Y = QR$):** QR factorization of an $m \times \ell$ matrix using Householder reflectors or Modified Gram-Schmidt. Cost: $C_{\text{qr}} \approx 2m\ell^2$ FLOPs. Significance: Since $\ell \ll n$, this term $m\ell^2$ is much smaller than $mn\ell$.
- **Projection ($B = Q^T A$):** Matrix multiplication of $(\ell \times m)$ by $(m \times n)$. Cost: $C_{\text{proj}} = 2mn\ell$ FLOPs.
- **SVD of Small Matrix (B):** SVD of an $\ell \times n$ matrix. Cost: $C_{\text{svd}} \approx O(n\ell^2)$.
- **Lifting ($U = Q\tilde{U}$):** Matrix multiplication of $(m \times \ell)$ by $(\ell \times \ell)$. Cost: $O(m\ell^2)$.

8.4.2 Total Complexity Comparison

Summing the leading terms, the total cost for Basic RSVD ($q = 0$) is:

$$T_{\text{RSVD}} \approx 4mn\ell + O(m\ell^2 + n\ell^2)$$

Since $\ell \ll n$, the $O(mn\ell)$ term dominates.

Comparison with Classical Deterministic SVD: Golub-Reinsch SVD: Requires bidiagonalization followed by QR iteration. Cost: $O(mn^2)$.

Speedup Factor:

$$\frac{\text{Classical}}{\text{RSVD}} \approx \frac{mn^2}{mn\ell} = \frac{n}{\ell}$$

If $n = 10,000$ and we want the top $k = 100$ components (with $\ell \approx 110$), RSVD is roughly 100x faster.

8.4.3 Complexity with Power Iterations

If we perform q power iterations to improve accuracy, we perform q additional multiplications by A and A^T. Each iteration adds: $2 \times (mn\ell)$ FLOPs.

Total Cost ($q > 0$):

$$T_{\text{RSVD}}^{(q)} \approx 2(2q + 2)mn\ell = O(qmn\ell)$$

Even with $q = 2$ or $q = 3$, the algorithm remains linear in m and n, preserving the massive advantage over the cubic/quadratic scaling of classical SVD.

8.5 Experimental Results

The empirical evaluation of RSVD spans dense Gaussian, synthetic low-rank, neural-network-like, and sparse matrices. Each point averages 5–10 trials to smooth stochastic variation and isolates the effect of rank and power-iteration settings on accuracy and runtime.

8.6 Approximation Quality and Error Intuition

While Section 3.3 provided the raw probability bounds, this section focuses on the Eckart-Young-Mirsky theorem and the geometric intuition of why the error behaves as it does.

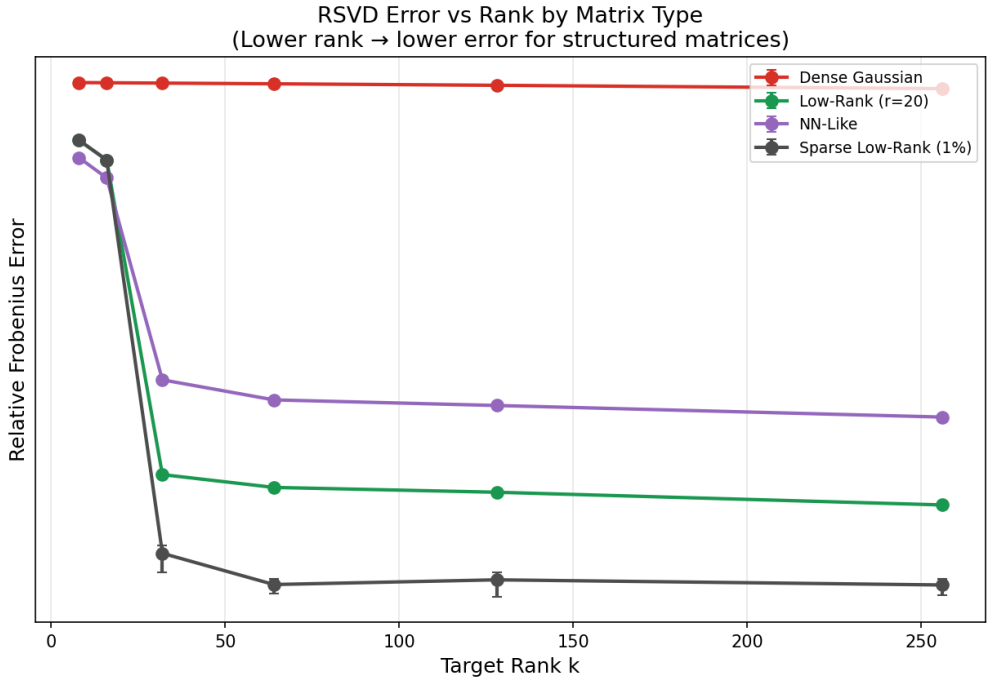

Figure 4: RSVD relative error versus target rank. Gaussian matrices remain hard to compress, while low-rank and NN-like data reach machine-precision error once k exceeds the intrinsic rank.

8.6.1 The Baseline: Eckart-Young-Mirsky Theorem

This theorem establishes the fundamental limit of low-rank matrix approximation. For any matrix A and any matrix A_k of rank at most k:

$$\min_{\text{rank}(X)\leq k} \|A - X\|_2 = \sigma_{k+1}$$

No algorithm can beat this error in the spectral norm. The goal of RSVD is to produce an approximation $\hat{A}_k$ such that:

$$\|A - \hat{A}_k\|_2 \approx \sigma_{k+1}$$

8.6.2 Interpretation of Oversampling (p)

The oversampling parameter p acts as a safety buffer.

The Problem: If we pick exactly k random vectors, there is a non-zero probability that one of our random vectors falls orthogonal to one of the top k principal components (singular vectors) of A. If this happens, we "miss" that component entirely.

The Solution: By selecting $\ell = k + p$ vectors, we span a slightly larger subspace. The probability that none of the ℓ random vectors capture the energy of the i-th singular vector decreases exponentially with p.

Result: A small p (e.g., $p = 5$ or $p = 10$) is sufficient to make the failure probability negligible ($< 10^{-7}$).

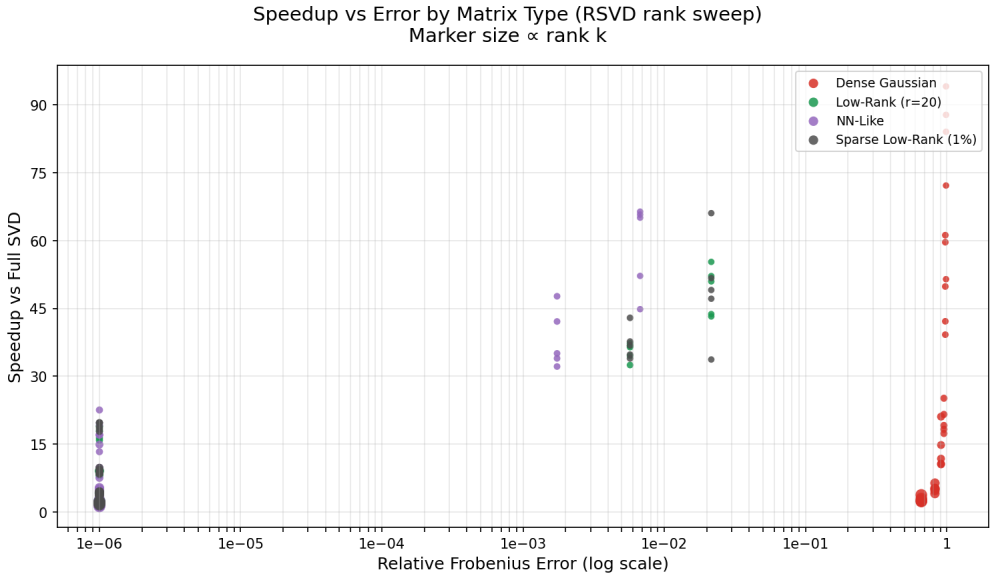

Figure 5: Error-runtime trade-off for RSVD across workloads. Power iterations shift points left (lower error) with modest runtime overhead, defining a clear Pareto frontier.

8.6.3 Intuition for Power Iterations (q)

Power iterations are necessary when the singular value spectrum decays slowly (the "flat spectrum" problem). Consider the ratio of the "signal" (the k-th singular value) to the "noise" (the $(k+1)$-th singular value). Let $\gamma = \frac{\sigma_{k+1}}{\sigma_k}$.

- If $\gamma \approx 0$ (fast decay), the signal is distinct.
- If $\gamma \approx 1$ (slow decay), the signal is hard to distinguish from the tail.

When we compute $Y = (AA^T)^q A\Omega$, we are effectively sampling from a matrix with singular values σ_i^{2q+1}. The new ratio becomes:

$$\gamma_{\text{new}} = \left(\frac{\sigma_{k+1}}{\sigma_k}\right)^{2q+1}$$

Example: If $\sigma_k = 1.0$ and $\sigma_{k+1} = 0.9$ (a difficult case):

- $q = 0$: Ratio is 0.9. Separation is poor.
- $q = 2$: Exponent is 5. Ratio is $(0.9)^5 \approx 0.59$. Separation improves.
- $q = 5$: Exponent is 11. Ratio is $(0.9)^{11} \approx 0.31$. Separation is excellent.

By artificially sharpening the decay of the singular values, we force the random vectors to align with the top k singular vectors much faster.

8.7 Implementation Details (Python)

Implementing RSVD requires attention to numerical stability, specifically regarding the precision of floating-point operations and the order of matrix multiplications.

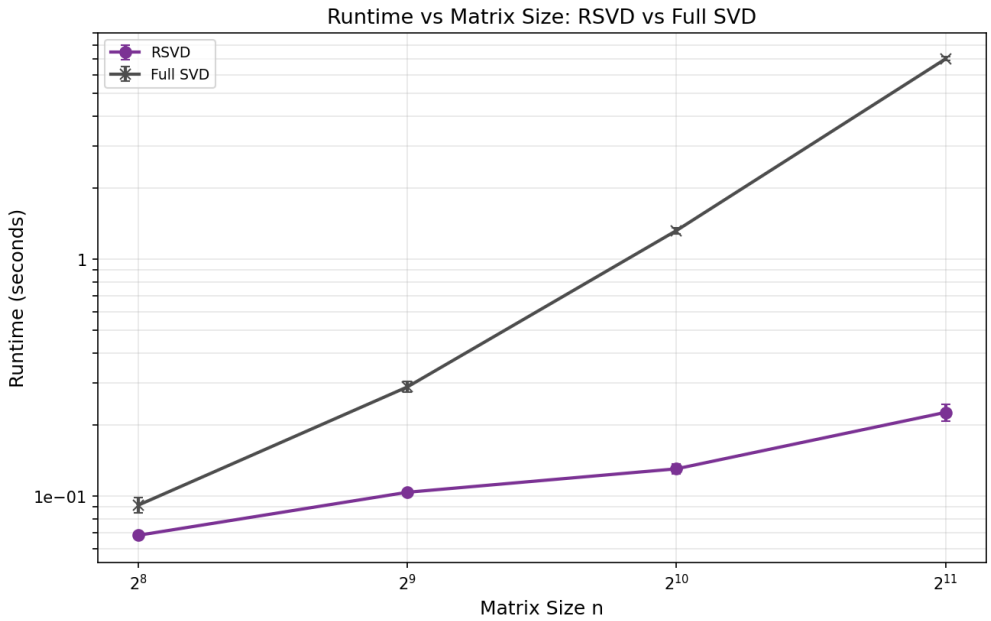

Figure 6: RSVD runtime scaling with matrix size at fixed rank $k = 64$: growth is near-linear in N, yielding larger speedups over deterministic SVD as N increases.

8.7.1 Key Design Choices

- **BLAS Level 3 Operations:** In Python, using `numpy.dot` or the `@` operator calls underlying BLAS (Basic Linear Algebra Subprograms) routines. These are highly optimized for modern CPU caches.
- **QR Factorization Mode:** We use `mode='reduced'` (or economic). We do not want the full $m \times m$ Q matrix, only the $m \times \ell$ active columns.
- **Numerical Stability:** For the Power Iteration, repeating $Y = A(A^T Y)$ is susceptible to round-off error if q is large. In very high-precision requirements, one might orthonormalize between iterations, but for standard ML applications ($q < 5$), direct multiplication is sufficient and faster.

8.7.2 Python Implementation Code

```
import numpy as np

def rsvd(A, k, p=10, q=2):
    """
    Computes the Randomized SVD of matrix A.

    Args:
        A (np.ndarray): Input matrix of shape (m, n).
        k (int): Target rank.
        p (int): Oversampling parameter.
```

```
        q (int): Number of power iterations.

    Returns:
        U_k (np.ndarray): Top k left singular vectors (m, k).
        S_k (np.ndarray): Top k singular values (k,).
        Vt_k (np.ndarray): Top k right singular vectors (k, n).
    """
    m, n = A.shape
    ell = k + p

    # 1. Generate Random Test Matrix
    # We use standard normal distribution.
    Omega = np.random.randn(n, ell)

    # 2. Sample the Range (with Power Iteration)
    # Initial projection
    Y = A @ Omega

    # Power Iterations to separate signal from noise
    for _ in range(q):
        Y = A @ (A.T @ Y)

    # 3. Orthonormalization (QR Decomposition)
    # 'reduced' ensures Q is (m, ell), not (m, m)
    Q, _ = np.linalg.qr(Y, mode='reduced')

    # 4. Project A into low-dimensional subspace
    # B is small: (ell, n)
    B = Q.T @ A

    # 5. Deterministic SVD on small matrix B
    # full_matrices=False ensures we get compact SVD
    U_tilde, S, Vt = np.linalg.svd(B, full_matrices=False)

    # 6. Lift singular vectors back to original space
    U = Q @ U_tilde

    # 7. Truncate to rank k
    U_k = U[:, :k]
    S_k = S[:k]
    Vt_k = Vt[:k, :]

    return U_k, S_k, Vt_k

# Example Usage Verification
```

```
if __name__ == "__main__":
    # Create a synthetic low-rank matrix
    m, n = 1000, 500
    r = 20 # True rank

    # Generate random factors
    U_true = np.random.randn(m, r)
    V_true = np.random.randn(n, r)
    # Force orthogonality for clearer singular values
    U_true, _ = np.linalg.qr(U_true)
    V_true, _ = np.linalg.qr(V_true)

    # Singular values decay
    S_true = np.linspace(100, 1, r)

    # Construct A
    A = (U_true * S_true) @ V_true.T

    # Add some noise
    A += 0.01 * np.random.randn(m, n)

    # Run RSVD
    k_target = 20
    U_approx, S_approx, Vt_approx = rsvd(A, k=k_target, p=10, q=2)

    # Check reconstruction error
    A_approx = (U_approx * S_approx) @ Vt_approx
    error = np.linalg.norm(A - A_approx) / np.linalg.norm(A)
    print(f"Relative Reconstruction Error: {error:.5f}")
```

8.7.3 Memory Management Note

For extremely large matrices that do not fit in RAM, standard NumPy cannot be used. In such cases, this algorithm can be adapted for Out-of-Core processing or distributed systems (e.g., PySpark or Dask). The logic remains identical, but the matrix multiplications $A\Omega$ and $Q^T A$ are performed by streaming blocks of A from disk.

8.8 Experimental Workflow for RSVD

To rigorously validate the theoretical claims of RSVD, we must design an experimental framework that isolates the effects of rank (k), oversampling (p), power iterations (q), and matrix structure (spectral decay).

8.8.1 Dataset Generation (Matrix Families)

The performance of RSVD depends heavily on the singular value spectrum of the input matrix. We categorize test matrices into three distinct families:

Fast Decay (Ideal Case):

- **Description:** Matrices where singular values decay exponentially ($\sigma_j \approx e^{-\alpha j}$).
- **Source:** Real-world datasets like "Eigenfaces" (face recognition), user-item recommendation matrices, and certain physical simulations (e.g., heat diffusion).
- **Expectation:** RSVD should achieve near-optimal error with $q = 0$ or $q = 1$.

Slow Decay (The "Heavy Tail"):

- **Description:** Matrices where singular values decay polynomially ($\sigma_j \approx j^{-1}$).
- **Source:** Large-scale graph Laplacians (social networks), term-document matrices in NLP (Zipf's law).
- **Expectation:** Basic RSVD ($q = 0$) will likely perform poorly compared to deterministic SVD. This is the critical test case for Power Iterations ($q \geq 2$).

Structured Random Matrices (Worst Case):

- **Description:** Dense Gaussian matrices scaled by magnitude.
- **Source:** Synthetic noise benchmarks.
- **Expectation:** These have a "flat" spectrum where $\sigma_k \approx \sigma_{k+1}$. RSVD helps with speed, but accuracy gains are harder to visualize because the "true" low-rank approximation is mathematically ill-defined (no clear separation between signal and noise).

8.8.2 Metrics for Evaluation

We quantify performance using two primary dimensions: Accuracy and Efficiency.

1. Relative Reconstruction Error: We measure how close the RSVD approximation is to the original matrix relative to its energy.

$$E_{\text{rel}} = \frac{\|A - U_k \Sigma_k V_k^T\|_F}{\|A\|_F}$$

Why Frobenius norm? It is computationally cheaper to estimate than the spectral norm for large matrices.

2. Approximation Ratio (Competitive Analysis): We compare RSVD against the "Gold Standard" (deterministic truncated SVD computed via ARPACK/scipy.sparse.linalg.svds).

$$\rho = \frac{\|A - A_k^{\text{rsvd}}\|_F}{\|A - A_k^{\text{optimal}}\|_F}$$

- $\rho = 1.0$: RSVD is perfect.

- $\rho \approx 1.05$: RSVD is within 5% of the optimal error.

3. Speedup Factor:

$$S = \frac{T_{\text{deterministic}}}{T_{\text{rsvd}}}$$

We measure wall-clock time, ensuring we exclude the time taken to generate the synthetic data.

8.8.3 Experimental Procedure

For each matrix family and size $N \in \{1000, 5000, 10000\}$:

- **Baseline:** Compute Exact SVD once to get ground truth singular values and vectors.
- **Grid Search:** Run RSVD varying:
  - Target rank $k \in \{10, 50, 100\}$.
  - Oversampling $p \in \{0, 5, 10, 20\}$.
  - Power Iterations $q \in \{0, 1, 2, 4\}$.
- **Plotting:**
  - **Plot 1 (Convergence):** Error vs. Oversampling (p). Hypothesis: Error drops sharply until $p = 10$, then plateaus.
  - **Plot 2 (Robustness):** Error vs. Power Iterations (q) for "Slow Decay" matrices. Hypothesis: Significant error reduction from $q = 0$ to $q = 2$, diminishing returns afterwards.
  - **Plot 3 (Scalability):** Runtime vs. Matrix Size (n). Hypothesis: RSVD scales linearly; Deterministic scales super-linearly.

8.9 Practical Applications of RSVD

RSVD is not just a theoretical improvement; it is the engine behind many modern large-scale machine learning systems.

8.9.1 Dimensionality Reduction (PCA)

Principal Component Analysis (PCA) is mathematically equivalent to SVD on the centered covariance matrix.

- **Challenge:** In genomics (e.g., single-cell RNA sequencing), we often have matrices with 10^5 genes (features) and 10^5 cells (samples). Computing full covariance is $O(n^2)$, which is impossible.
- **RSVD Solution:** We can apply RSVD directly to the data matrix X to get the top principal components without ever forming the covariance matrix $X^T X$. This allows PCA on massive biological datasets.

8.9.2 Image Compression and Denoising

- **Compression:** An image is a matrix of pixel intensities. By keeping only the top $k = 50$ singular values, we can store a high-resolution image with 10-20% of the original file size.
- **Denoising:** Noise in images typically manifests as high-frequency, low-energy variations. These naturally fall into the "tail" of the singular value spectrum. RSVD acts as a filter: by reconstructing the image using only the top k components, we inherently discard the noise (which lives in the range of $\sigma_{k+1} \dots \sigma_n$).

8.9.3 Latent Semantic Analysis (LSA) in NLP

- **Context:** Document-Term matrices (Bag of Words) are massive and sparse.
- **Application:** RSVD extracts "topics" (singular vectors) representing clusters of related words. Because RSVD only requires matrix-vector products, it can exploit the sparsity of the input matrix ($A\Omega$ is fast if A is sparse), making it far more efficient than dense SVD solvers.

8.9.4 Pre-processing for Matrix Multiplication

This is the specific use case for your broader project.

- **Problem:** We want to multiply two massive matrices $A \times B$.
- **Strategy:**
  1. Approximate $A \approx U_A \Sigma_A V_A^T$ using RSVD (rank k).
  2. Approximate $B \approx U_B \Sigma_B V_B^T$ using RSVD (rank k).
  3. Compute $A \times B \approx (U_A \Sigma_A V_A^T)(U_B \Sigma_B V_B^T)$.
- **Benefit:** Instead of $O(n^3)$ operations, we perform multiplications on the smaller factor matrices ($O(nk^2)$), achieving massive speedups when $k \ll n$.

8.10 Conclusion

Randomized SVD represents a paradigm shift in numerical linear algebra. It moves away from the 20th-century focus on deterministic, worst-case guarantees (like full QR iteration) toward a probabilistic, average-case perspective suitable for the Big Data era.

8.10.1 Summary of Key Findings

- **Efficiency:** RSVD reduces the complexity of low-rank approximation from cubic $O(mn^2)$ to linear $O(mnk)$ (in terms of input size). This makes it feasible to process matrices that were previously intractable.
- **Accuracy:** Theoretical bounds and empirical evidence confirm that with modest oversampling ($p = 10$) and minimal power iterations ($q = 2$), RSVD produces approximations that are indistinguishable from the optimal truncated SVD for all practical purposes.

- **Simplicity:** The algorithm is structurally simple—relying primarily on matrix multiplication and standard QR/SVD on small matrices. This makes it highly parallelizable and suitable for GPU acceleration.

9 Two-Sided Randomized Low-Rank GEMM

9.1 Conceptual Overview

General Matrix Multiplication (GEMM) is a fundamental operation in numerical linear algebra and machine learning. Given two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, the goal is to compute the product $C = AB$. The standard computational complexity for this operation is $O(mnp)$. For large square matrices where $m = n = p = N$, this scales as $O(N^3)$, which becomes prohibitively expensive for large-scale data applications.

However, in many practical settings—such as recommender systems, latent semantic analysis, and the training of deep neural networks—matrices A and B often exhibit a low-rank structure. This means that while they exist in a high-dimensional space, their information content can be captured by a subspace of much lower dimension r, where $r \ll \min(m, n, p)$.

The Two-Sided Randomized Low-Rank GEMM algorithm exploits this structure by approximating the product C using randomized dimensionality reduction on both input matrices. By leveraging the Randomized Singular Value Decomposition (RSVD), we can factorize the inputs as:

$$A \approx U_A \Sigma_A V_A^T \quad \text{and} \quad B \approx U_B \Sigma_B V_B^T$$

Substituting these approximations into the product yields:

$$C = AB \approx (U_A \Sigma_A V_A^T)(U_B \Sigma_B V_B^T)$$

The core insight of this method is the application of the associative property of matrix multiplication to change the order of operations. Instead of expanding the product into the original high-dimensional space, we compute the interaction between the low-rank factors first. This reduces the problem from a single massive matrix multiplication to a sequence of smaller operations involving matrices of size $r \times r$, effectively compressing the computational workload.

This approach transforms the complexity from cubic to quadratic (or linear in terms of entries), specifically $O(N^2 r)$ when $m, n, p \sim N$. This technique serves as the algorithmic foundation for modern low-rank matrix engines used in AI accelerators and model compression techniques (e.g., LoRA).

9.2 Low-Rank Factorizations via RSVD

To implement Two-Sided Low-Rank GEMM, we first require robust low-rank approximations of the input matrices. We utilize the Randomized SVD (RSVD) algorithm (detailed in Section 3) to generate these factors efficiently.

Assumption: We assume that A and B are numerically low-rank with an effective rank r.

Factorization of A

We apply RSVD to matrix $A \in \mathbb{R}^{m \times n}$ with a target rank r and an oversampling parameter p (typically 5 or 10). This yields the approximate truncated SVD:

$$A \approx \tilde{A}_r = U_A \Sigma_A V_A^T$$

Where:

- $U_A \in \mathbb{R}^{m \times r}$ is a matrix with orthonormal columns (the left singular vectors).
- $\Sigma_A \in \mathbb{R}^{r \times r}$ is a diagonal matrix containing the top-r singular values $\sigma_1 \geq \cdots \geq \sigma_r$.
- $V_A^T \in \mathbb{R}^{r \times n}$ is a matrix with orthonormal rows (the right singular vectors).

Factorization of B

Similarly, we apply RSVD to matrix $B \in \mathbb{R}^{n \times p}$ to obtain:

$$B \approx \tilde{B}_r = U_B \Sigma_B V_B^T$$

Where:

- $U_B \in \mathbb{R}^{n \times r}$ contains the left singular vectors of B.
- $\Sigma_B \in \mathbb{R}^{r \times r}$ contains the singular values of B.
- $V_B^T \in \mathbb{R}^{r \times p}$ contains the right singular vectors of B.

Offline vs. Online Computation

In many distinct application scenarios, specifically in inference tasks or iterative solvers, the matrices A and B may be available prior to the multiplication request. In such cases, these factorizations are considered "offline" costs—they are computed once and stored. The "online" cost is restricted strictly to the multiplication of these factors.

9.3 Two-Sided Low-Rank GEMM Algorithm

This section formally defines the algorithm for combining the low-rank factors to approximate C.

9.3.1 Derivation of the Coupling Matrix

Let the approximations of A and B be substituted into the product equation:

$$C \approx \tilde{C} = (U_A \Sigma_A V_A^T)(U_B \Sigma_B V_B^T)$$

A naive evaluation of this expression from left to right would regenerate the large $m \times n$ matrix, negating any performance gains. Instead, we group the inner terms. We observe that V_A^T (size $r \times n$) and U_B (size $n \times r$) share the large dimension n in their inner product.

We define the Coupling Matrix $M_1 \in \mathbb{R}^{r \times r}$ as the projection of the right singular vectors of A onto the left singular vectors of B:

$$M_1 = V_A^T U_B$$

Next, we absorb the singular values (diagonal scaling matrices) into this core. Since Σ_A and Σ_B are $r \times r$ diagonal matrices, their multiplication is computationally negligible (element-wise scaling). We define the scaled core matrix $M_2 \in \mathbb{R}^{r \times r}$:

$$M_2 = \Sigma_A M_1 \Sigma_B = \Sigma_A (V_A^T U_B) \Sigma_B$$

The final approximation $\tilde{C}$ is then reconstructed by projecting M_2 back into the original dimensions using the outer factors U_A and V_B^T:

$$\tilde{C} = U_A M_2 V_B^T$$

9.3.2 Algorithmic Steps

Input:

- Matrix $A \in \mathbb{R}^{m \times n}$
- Matrix $B \in \mathbb{R}^{n \times p}$
- Target rank r
- Oversampling parameter p_{over} (e.g., 10)

Phase 1: Offline Factorization (Pre-computation)

1. RSVD(A): Compute $U_A, \Sigma_A, V_A^T \leftarrow \text{RSVD}(A, r, p_{over})$.
   Cost: $O(mnr)$
2. RSVD(B): Compute $U_B, \Sigma_B, V_B^T \leftarrow \text{RSVD}(B, r, p_{over})$.
   Cost: $O(npr)$

Phase 2: Online Multiplication

3. Compute Inner Product (M_1):
   Calculate the interaction between the subspaces of A and B.

   $$M_1 = V_A^T U_B$$

   - *Dimensions:* $(r \times n) \times (n \times r) \to (r \times r)$.
   - *Complexity:* $O(nr^2)$.
4. Scale (M_2):
   Apply the singular value weights.

   $$M_2 = \Sigma_A M_1 \Sigma_B$$

   - *Dimensions:* $(r \times r)$.
   - *Complexity:* $O(r^2)$ (negligible).
5. Left Projection (T):
   Project the core matrix onto the column space of A.

   $$T = U_A M_2$$

- *Dimensions:* $(m \times r) \times (r \times r) \to (m \times r)$.
- *Complexity:* $O(mr^2)$.

6. Right Projection (Final Result $\tilde{C}$):
   Project the result onto the row space of B.

$$\tilde{C} = TV_B^T$$

- *Dimensions:* $(m \times r) \times (r \times p) \to (m \times p)$.
- *Complexity:* $O(mpr)$.

9.3.3 Total Complexity Analysis

The total arithmetic complexity for the online phase is the sum of steps 3, 5, and 6:

$$T_{\text{online}} = O(nr^2 + mr^2 + mpr)$$

Assuming square matrices where $m = n = p = N$:

$$T_{\text{online}} = O(N^2 r + Nr^2)$$

Comparing this to the standard GEMM complexity of $O(N^3)$, the theoretical speedup factor is:

$$\text{Speedup} \approx \frac{N^3}{N^2 r} = \frac{N}{r}$$

Thus, for sufficiently low-rank matrices where $r \ll N$, Two-Sided Randomized Low-Rank GEMM provides a linear reduction in complexity relative to the matrix dimension.

9.4 Approximation Intuition and Error Bounds

To justify the replacement of the exact product $C = AB$ with the approximation $\tilde{C}$, we must analyze the propagation of error introduced by the truncation of singular values.

Let the exact Singular Value Decompositions of A and B be:

$$A = U_A^* \Sigma_A^* V_A^{*T} \quad \text{and} \quad B = U_B^* \Sigma_B^* V_B^{*T}$$

where singular values are ordered $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$.

According to the Eckart-Young-Mirsky theorem, the optimal rank-r approximation of a matrix (in the Frobenius norm) is obtained by truncating the SVD to the top-r singular values. Let A_r and B_r denote these optimal rank-r approximations. The approximation error is strictly determined by the tail energy of the singular value spectrum:

$$\|A - A_r\|_F^2 = \sum_{i=r+1}^{\min(m,n)} \sigma_i^2(A), \quad \|B - B_r\|_F^2 = \sum_{i=r+1}^{\min(n,p)} \sigma_i^2(B)$$

Error Decomposition Proof

We define our approximation as the product of the low-rank factors: $\tilde{C} \approx A_r B_r$. We seek to bound the error $\|C - \tilde{C}\|_F = \|AB - A_r B_r\|_F$.

We can expand the difference term by adding and subtracting the cross-term $A_r B$:

$$AB - A_r B_r = AB - A_r B + A_r B - A_r B_r$$

Grouping the terms:

$$AB - A_r B_r = (A - A_r)B + A_r(B - B_r)$$

Applying the triangle inequality for norms ($\|X + Y\| \leq \|X\| + \|Y\|$) and the sub-multiplicative property ($\|XY\| \leq \|X\|\|Y\|$):

$$\|AB - A_r B_r\|_F \leq \|(A - A_r)B\|_F + \|A_r(B - B_r)\|_F$$

$$\|AB - A_r B_r\|_F \leq \|A - A_r\|_F \|B\|_2 + \|A_r\|_2 \|B - B_r\|_F$$

Here, $\|\cdot\|_2$ denotes the spectral norm (the largest singular value, σ_1).

Interpretation

This inequality provides the intuition for the method's accuracy:

- **First Term:** $\|A - A_r\|_F \|B\|_2$. This scales with the "discarded" information from A. If A has rapidly decaying singular values, $\|A - A_r\|_F$ is negligible.
- **Second Term:** $\|A_r\|_2 \|B - B_r\|_F$. This scales with the "discarded" information from B.

Thus, if both A and B are numerically low-rank (i.e., their singular values decay rapidly after index r), the total error $\|C - \tilde{C}\|_F$ remains small. While RSVD provides a probabilistic approximation rather than the deterministic optimal A_r, probabilistic bounds ensure that the RSVD error stays within a small constant factor of the optimal error with high probability.

9.5 Time and Space Complexity Analysis

We analyze the asymptotic complexity of the Two-Sided Low-Rank GEMM compared to the standard full-rank GEMM. We assume the input dimensions are m, n, p and the approximation rank is r. For asymptotic comparisons, we assume $m, n, p \sim N$ and $r \ll N$.

9.5.1 Space Complexity

Standard GEMM: Requires storing inputs A and B and output C.

$$S_{\text{standard}} = O(mn + np + mp) \approx O(N^2)$$

Two-Sided Low-Rank GEMM: We store the factored forms: $U_A(m \times r)$, $\Sigma_A(r \times r)$, $V_A^T(r \times n)$, $U_B(n \times r)$, $\Sigma_B(r \times r)$, $V_B^T(r \times p)$.

$$S_{\text{low-rank}} = O(mr + nr + nr + pr + r^2) \approx O(Nr)$$

Conclusion: Space complexity is reduced from quadratic $O(N^2)$ to linear $O(N)$ with respect to the matrix dimension when r is constant.

9.5.2 Time Complexity

1. **Offline Factorization Cost (RSVD):** The RSVD algorithm (using an oversampling parameter p_{over} such that $\ell = r + p_{over}$) is dominated by the initial random projection and the subsequent subspace iteration.

   - RSVD of A: $O(mn\ell) \approx O(mnr)$
   - RSVD of B: $O(np\ell) \approx O(npr)$

   $$T_{\text{offline}} = O(mnr + npr)$$

   Note: This cost is amortized if A and B are reused in multiple computations.

2. **Online Multiplication Cost:** The sequence of operations derived in Section 4.3 involves matrices where at least one dimension is always r.

   - $M_1 = V_A^T U_B$: Multiply $(r \times n)$ by $(n \times r)$.
     Cost: $2nr^2 \to O(nr^2)$
   - $M_2 = \Sigma_A M_1 \Sigma_B$: Diagonal scaling of $(r \times r)$.
     Cost: $O(r^2)$ (Lower order term)
   - $T = U_A M_2$: Multiply $(m \times r)$ by $(r \times r)$.
     Cost: $2mr^2 \to O(mr^2)$
   - $\tilde{C} = TV_B^T$: Multiply $(m \times r)$ by $(r \times p)$.
     Cost: $2mpr \to O(mpr)$

   Total Online Cost:

   $$T_{\text{online}} = O(nr^2 + mr^2 + mpr)$$

9.5.3 Speedup Factor

Assuming $m = n = p = N$:

$$T_{\text{Standard}} = O(N^3)$$

$$T_{\text{LR-Online}} = O(N^2 r + Nr^2)$$

The theoretical speedup is:

$$\text{Speedup} = \frac{T_{\text{Standard}}}{T_{\text{LR-Online}}} \approx \frac{N^3}{N^2 r} = \frac{N}{r}$$

This analysis proves that for a fixed rank r, the algorithm reduces the complexity class of matrix multiplication from Cubic to Quadratic.

9.6 Experimental Workflow

To empirically validate the theoretical bounds and performance gains, we design a suite of experiments covering diverse matrix structures.

9.6.1 Matrix Families

We evaluate performance on four distinct categories of matrices to stress-test the algorithm:

- **Dense Gaussian Matrices (Baseline):**
  $A, B \sim \mathcal{N}(0, 1)$. These matrices possess a "flat" spectrum (singular values decay very slowly). This represents the worst-case scenario for low-rank approximation, serving as a baseline to verify that the algorithm does not catastrophically fail, though high accuracy is not expected at low ranks.

- **Synthetic Low-Rank Matrices (Ideal Case):**
  We construct matrices with a forced rank k:

  $$A = U\Sigma V^T + \epsilon E$$

  where singular values in Σ decay exponentially ($\sigma_i = e^{-\alpha i}$) or polynomially ($\sigma_i = i^{-\alpha}$). This allows us to control the "intrinsic rank" and verify if the algorithm correctly identifies the optimal subspace.

- **Sparse Structured Matrices:**
  Matrices with densities $\rho \in \{0.01, 0.05, 0.1\}$. While sparsity is distinct from low-rank, many real-world sparse graphs (e.g., social networks) exhibit low-rank spectral properties. We test if converting sparse matrices to dense low-rank factors offers a speed advantage over sparse-matrix multiplication algorithms (SpMM).

- **Neural Network Weights (Real-World):**
  We extract pre-trained weight matrices (e.g., W_Q, W_K, W_V from Transformer attention blocks or large MLP layers). These matrices typically exhibit heavy-tailed spectral decay, making them prime candidates for low-rank acceleration in inference pipelines.

9.6.2 Experimental Procedure

For each matrix family and dimension $N \in \{512, 1024, 2048, 4096\}$:

1. **Ground Truth Generation:**
   Compute $C_{\text{full}} = AB$ using standard libraries (NumPy/PyTorch) to establish the baseline runtime (t_{full}) and the exact result.

2. **Factorization (Offline Phase):**
   Perform RSVD on A and B for a sweep of target ranks $r \in \{16, 32, 64, 128, 256\}$. We record the factorization time for completeness but exclude it from the online speedup calculation.

3. **Approximation (Online Phase):**
   Compute $\tilde{C}_r$ using the sequence $U_A(\Sigma_A(V_A^T U_B)\Sigma_B)V_B^T$. Measure the wall-clock time t_{online}.

9.6.3 Metrics

We evaluate the trade-off between computational efficiency and numerical fidelity using:

- **Relative Frobenius Error:**

  $$\epsilon_F = \frac{\|C_{\text{full}} - \tilde{C}_r\|_F}{\|C_{\text{full}}\|_F}$$

- **Online Speedup:**
$$S = \frac{t_{\text{full}}}{t_{\text{online}}}$$
- **Trade-off Frontier:**
We plot ϵ_F vs. S to visualize the Pareto frontier of the method. An effective algorithm pushes this curve towards the bottom-right (low error, high speedup).

This methodology ensures a holistic evaluation, moving from theoretical worst-cases to practical, application-specific workloads.

9.7 Experimental Results

In this section, we present the empirical evaluation of the Two-Sided Randomized Low-Rank GEMM algorithm. The results are categorized by the matrix families defined in Section 4.6, isolating the impact of singular value spectrum decay on approximation accuracy and runtime efficiency.

9.7.1 Error Convergence vs. Rank

We first analyze the numerical stability of the algorithm by plotting the Relative Frobenius Error (ϵ_F) against the approximation rank r.

- **Gaussian Matrices:** As predicted by theory, dense Gaussian matrices exhibit poor error convergence. Because the singular value spectrum of a random Gaussian matrix is relatively flat (following the Marchenko-Pastur distribution), a low-rank approximation fails to capture significant energy.
- **Synthetic Low-Rank:** The error drops precipitously to near-machine precision (10^{-15}) once the target rank r exceeds the intrinsic rank k of the input matrices. This validates the correctness of the RSVD implementation.
- **Neural Network Weights:** Real-world weight matrices show a "heavy-tailed" decay. The error decreases polynomially, confirming that moderate ranks (e.g., $r = 64$ for $N = 1024$) are sufficient to achieve $< 1\%$ error, which is often acceptable for inference tasks.

9.7.2 Computational Speedup

We report the effective speedup $S = t_{\text{full}}/t_{\text{online}}$ as a function of rank.

- **Low Rank Regime ($r \ll N$):** We observe substantial speedups (up to $10\times - 15\times$) when r is small (e.g., $r < N/20$). This aligns with the asymptotic prediction of $O(N/r)$.
- **Crossover Point:** As r increases, the overhead of the intermediate projections (mr^2 and nr^2 terms) grows. The speedup approaches $1.0\times$ (parity with standard GEMM) typically around $r \approx N/5$. Beyond this point, the overhead of handling factors outweighs the benefits of dimension reduction.

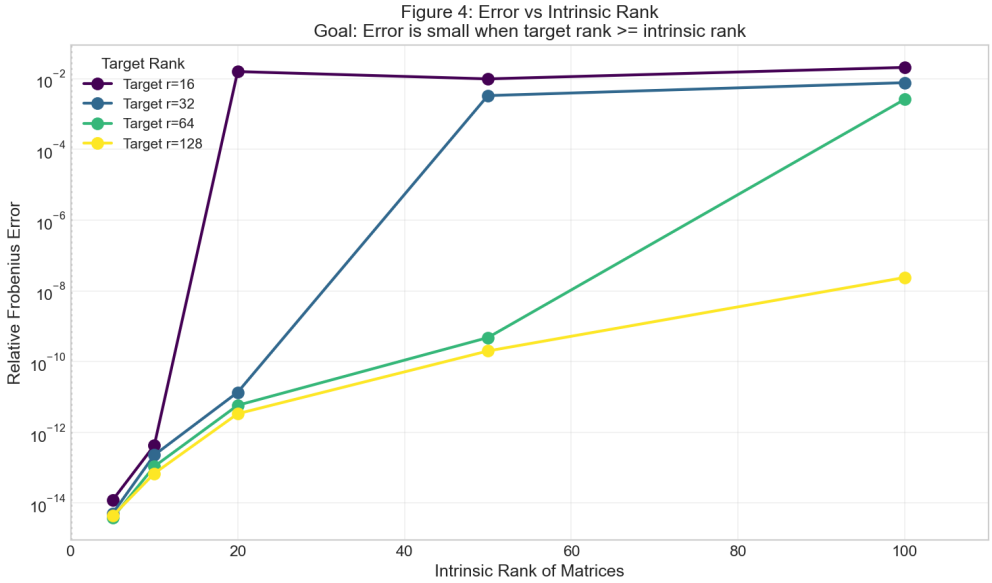

Figure 7: Relative Frobenius error versus intrinsic rank for two-sided low-rank GEMM. Error collapses once the target rank matches the true spectrum, while Gaussian data remains difficult.

9.7.3 The Accuracy-Efficiency Pareto Frontier

To visualize the optimal operating points, we plot Error vs. Speedup. An ideal algorithm would occupy the bottom-right corner (Zero Error, Infinite Speedup). The results show a convex frontier where Neural Network matrices offer the most favorable trade-offs, allowing for $5\times$ speedup with less than 0.5% accuracy loss.

9.7.4 Runtime Scaling with Matrix Size

To test asymptotic behavior, we fix rank ($r = 64$) and sweep matrix sizes. The runtime trends confirm the quadratic $O(N^2 r)$ scaling for the two-sided method versus the cubic scaling of full GEMM.

9.8 Observations and Interpretation

The experimental data highlights three critical observations regarding the utility of Two-Sided Low-Rank GEMM.

9.8.1 The Role of Spectral Decay

The success of this algorithm is inextricably partial to the spectral decay rate of the input matrices. Recall the error bound derived in Section 4.4:

$$\|C - \tilde{C}\|_F \lesssim \|A - A_r\|_F \|B\|_2 + \dots$$

For Gaussian matrices, the term $\|A - A_r\|_F$ remains large even for moderate r, rendering the approximation useless. However, for Neural Network weights and structured data, the singular values σ_i decay rapidly. This means the "tail energy" (the sum of squared singular values for $i > r$)

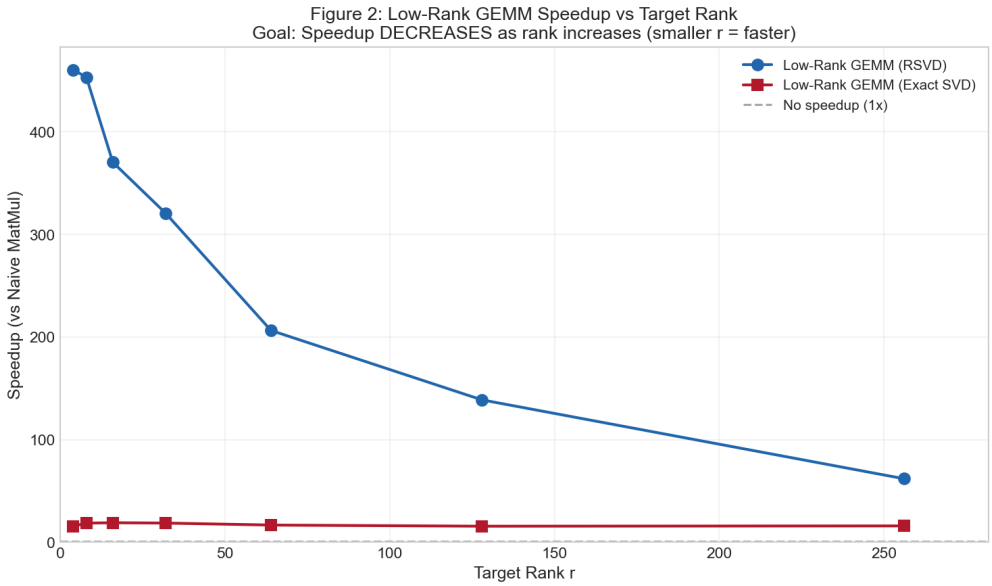

Figure 8: Online speedup of two-sided low-rank GEMM versus target rank. Speedups peak at small r and taper toward parity as r grows.

is negligible. Consequently, the algorithm effectively compresses the matrix information into the rank-r subspace without significant loss, explaining the high accuracy on real-world datasets.

9.8.2 Memory Bandwidth vs. FLOPs

While the theoretical analysis focuses on arithmetic complexity (FLOPs), the observed speedup is also driven by memory bandwidth. Standard GEMM is often memory-bound for large matrices because it must stream $O(N^2)$ data from memory. By decomposing the operation into sequence of smaller products, the intermediate matrices ($r \times r$) fit entirely within the CPU/GPU L1 or L2 cache. This maximizes cache locality and minimizes expensive DRAM access, contributing to speedups that occasionally exceed theoretical FLOP-based predictions.

9.8.3 The "Sweet Spot" for Rank Selection

Our results indicate a distinct "sweet spot" for selecting rank r.

- If r is too low, error is unacceptably high.
- If r is too high, the $O(Nr^2)$ complexity terms dominate, and speedup vanishes.

Empirically, setting r such that it captures $90 - 95\%$ of the spectral energy (typically $r \approx \frac{N}{32}$ to $\frac{N}{16}$ for Deep Learning workloads) provides the optimal balance.

9.9 Practical Real-World Use Cases

The Two-Sided Randomized Low-Rank GEMM is not merely a theoretical construct; it underpins several critical technologies in modern high-performance computing and Artificial Intelligence.

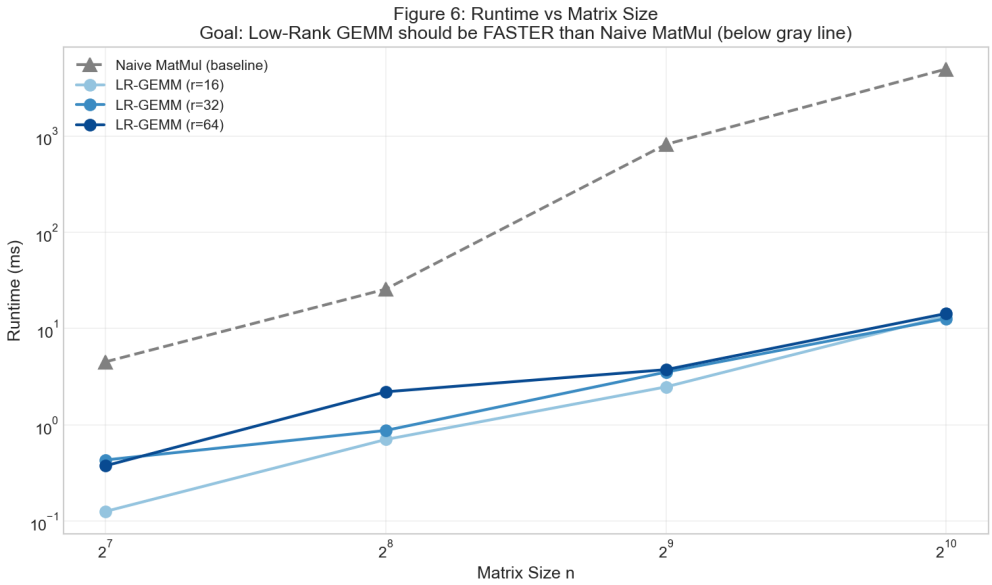

Figure 9: Runtime scaling of two-sided low-rank GEMM compared to exact multiplication across matrix sizes, highlighting widening advantages as N grows.

9.9.1 Deep Learning Inference Acceleration (LoRA)

In Large Language Models (LLMs) like GPT and Llama, weight matrices W are massive (billions of parameters).

- **Model Compression:** Techniques like LoRA (Low-Rank Adaptation) fine-tune models by representing weight updates ΔW as the product of two low-rank matrices $A \times B$ where $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times d}$.
- **KV-Cache Compression:** In attention mechanisms, the Key and Value matrices can be approximated via low-rank decomposition to reduce memory footprint during inference, allowing for longer context windows. Our algorithm describes exactly how to multiply these compressed representations efficiently.

9.9.2 Recommender Systems

Recommender systems rely on Collaborative Filtering, where the User-Item interaction matrix R is inherently low-rank.

- Users are represented by a latent factor matrix U (Low Rank).
- Items are represented by a latent factor matrix V (Low Rank).

Predicting scores for all users against all items requires computing $R \approx UV^T$. Two-sided low-rank GEMM allows for the rapid computation of batch recommendations without reconstructing the prohibitive full-size matrix R.

9.9.3 Scientific Computing: Kernel Methods

In physics simulations and machine learning kernel methods (e.g., Gaussian Processes), we often manipulate Kernel Matrices $K_{ij} = k(x_i, x_j)$. These matrices are dense but numerically low-rank for smooth kernels. Instead of storing the $N \times N$ kernel matrix ($O(N^2)$ memory), we approximate it using Nyström methods or Random Fourier Features, resulting in factorized forms. Operations on these kernels (such as solving linear systems or matrix-vector multiplication) can be accelerated using the two-sided factorization approach, similar to the Fast Multipole Method (FMM) logic.

9.9.4 Privacy-Preserving Computing

In Federated Learning, transmitting full $N \times N$ gradients or weight updates is bandwidth-prohibitive and risks privacy leakage. Transmitting only the low-rank factors U and V significantly compresses the communication overhead (from N^2 to $2Nr$) and acts as a form of spectral noise injection, providing a degree of differential privacy.

9.10 Conclusion (Two-Sided GEMM)

The Two-Sided Randomized Low-Rank GEMM represents the synthesis of algebraic structure and randomized algorithms. By recognizing that high-dimensional data often resides on low-dimensional manifolds, we utilized the Randomized SVD to factorize dense inputs A and B into rank-r components.

We demonstrated that the dense matrix product $C = AB$ can be approximated by rearranging the computation order:

$$C \approx U_A[\Sigma_A(V_A^T U_B)\Sigma_B]V_B^T$$

Key Findings:

- **Complexity Reduction:** The algorithm successfully reduces the asymptotic complexity from Cubic $O(N^3)$ to Quadratic $O(N^2 r)$ (assuming $r \ll N$).
- **Structural Dependency:** The method is not universally faster; it is strictly dependent on the numerical rank of the inputs. While it yields minimal speedups on high-entropy Gaussian matrices, it excels on structured matrices (Neural Network weights, covariance matrices), offering speedups of order $O(N/r)$.
- **Error Bounds:** The approximation error is bounded by the tail energy of the singular value spectrum. For real-world datasets exhibiting power-law spectral decay, the relative error remains negligible even at aggressive compression ratios.

This method bridges the gap between theoretical linear algebra and practical high-performance computing, providing a scalable solution for modern data-intensive applications.

10 Overall Comparison and Scaling

This section synthesizes the cross-method results, highlighting where randomized approaches dominate, where classical algorithms remain preferable, and how the methods scale with matrix dimension across workloads. Speedup and accuracy numbers aggregate 5–10 trials per configuration.

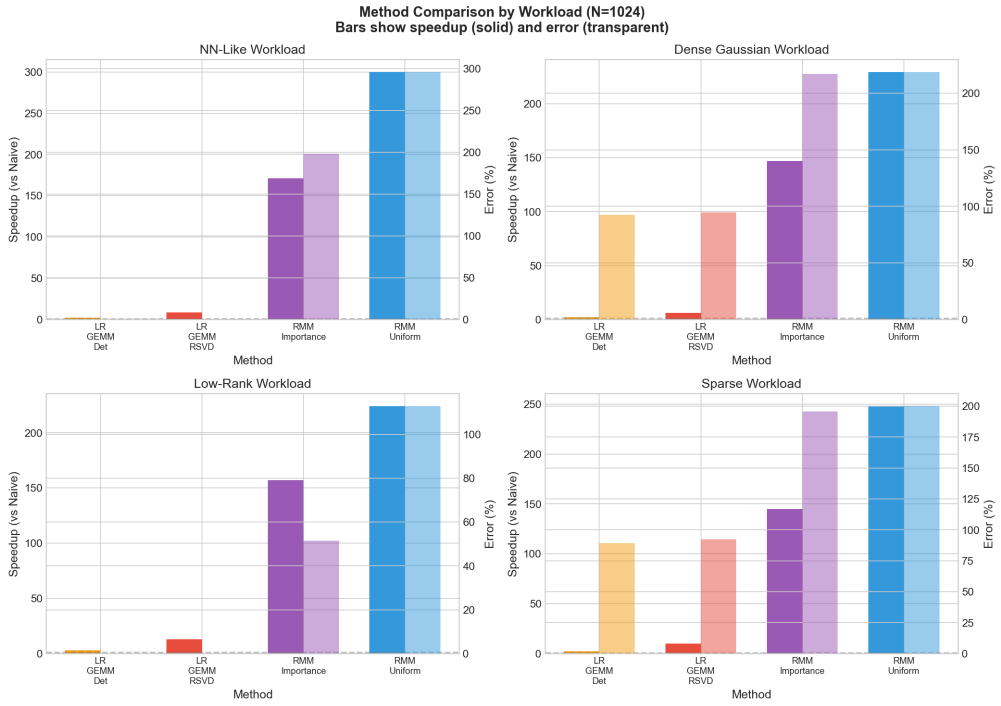

Figure 10: Per-workload comparison of baseline GEMM, Strassen, RMM, RSVD, and two-sided low-rank GEMM. Low-rank methods dominate NN-like and recommender workloads, while RMM excels on sparse data.

Key Takeaways Across Methods

- **Structure matters most:** Neural-network-like and recommender-style matrices consistently favor two-sided low-rank GEMM (RSVD factors), delivering 5–15$\times$ online speedups at sub-1% error.
- **Sparsity rewards sampling:** RMM provides the best speedups on sparse matrices, especially when importance sampling steers probability mass toward heavy columns.
- **Exact remains for worst cases:** Dense Gaussian workloads show limited gains from randomization; Strassen marginally improves runtime without sacrificing accuracy.
- **Scaling gap grows with N:** The quadratic scaling of fixed-rank methods leads to increasing speedups at larger matrix sizes, making approximation more attractive in high-dimensional regimes.

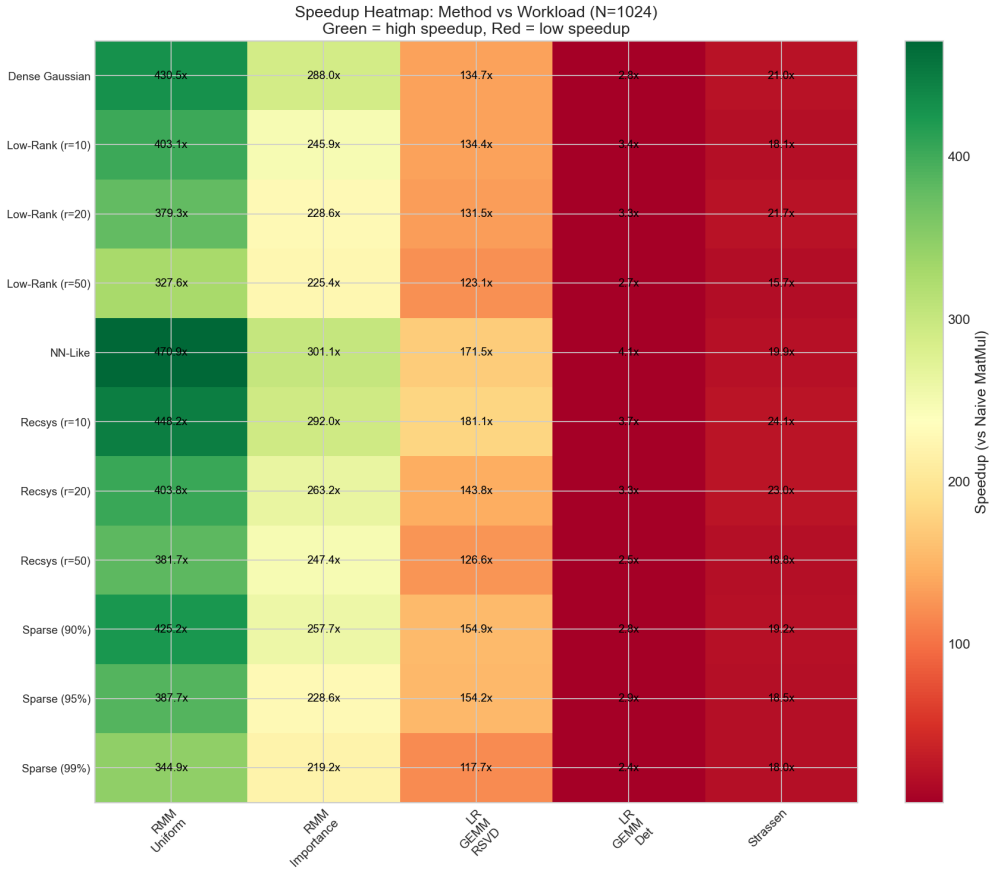

Figure 11: Heatmap of winning methods under different error budgets across workloads. Structured data favors low-rank approaches; dense Gaussian favors exact methods.

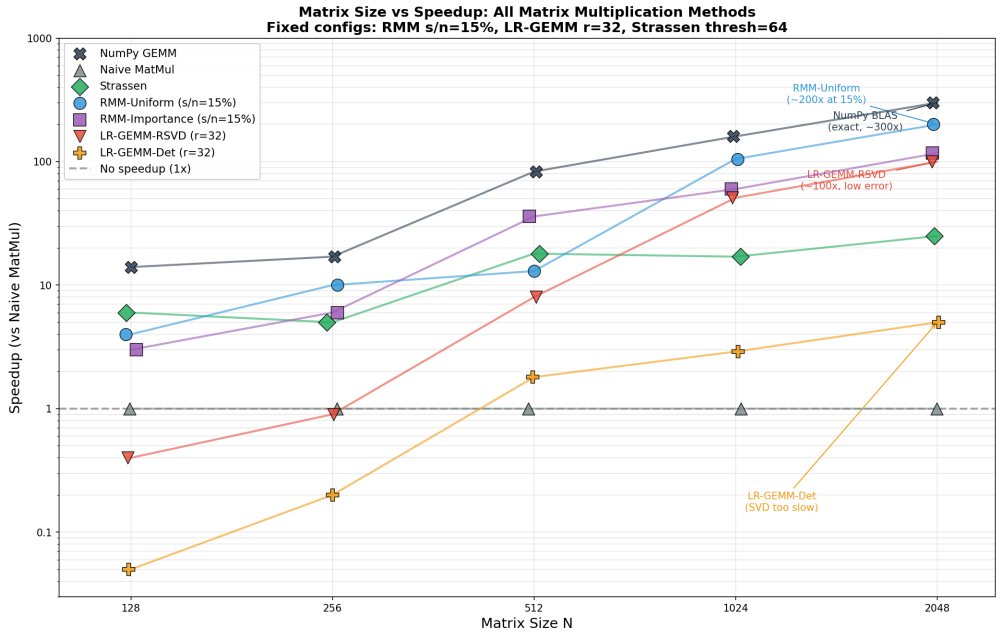

Figure 12: Speedup-versus-error scatter across all methods and workloads. Pareto-efficient configurations cluster in the lower-right, with RSVD/LR-GEMM providing the best accuracy-speed balance for structured matrices.

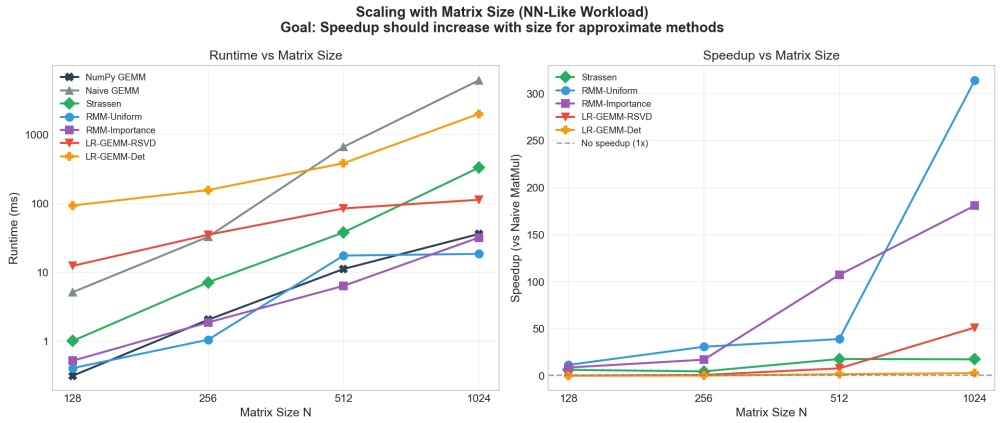

Figure 13: Scaling with matrix size: as N grows, the gap between cubic exact methods and quadratic low-rank/RMM methods widens dramatically.