Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.18576/amis/200406

Memory-Bound Scaling and the R–C++ Performance Crossover Threshold

Gharib M. Gharib[1], *Maha Alsaoudi*[2], *Jeireis Abudayyeh*[1], *Karim Adel*[3], *Mahmoud I. Gaber*[3], *and Mohamed A. Labeeb*[3,*]

[1] Department of Mathematics, Zarqa University, Zarqa, Jordan
[2] Department of Science, Applied Science Private University, Amman, Jordan
[3] Faculty of Artificial Intelligence, Egyptian Russian University, Cairo, Egypt

Received: 16 Jan. 2026, Revised: 4 Mar. 2026, Accepted: 22 Jun. 2026
Published online: 1 Jul. 2026

Abstract: In the current investigation, the performance crossover threshold of ordinary least squares (OLS) regression and frequency distribution is assessed for R, Rcpp, and C++ versions for large data analysis tasks. To assess the three algorithms, a benchmarking suite is designed which can validate the performance comparison of all implementations based on different data set sizes, from 3x105 to 108 records, in an Intel Core i7 environment. The framework allows measuring execution overheads, scalability characteristics, threading effects, and memory performance limitations. Experimental results show that the performance gap between R and C++ implementations mainly results from additional computation and statistics involved in R and not through foreign function interfaces. Furthermore, both OLS regression and frequency distribution implementations exhibit performance degradation because of the memory limit, rendering any improvements in threading ineffective and causing a drastic decline in performance crossover threshold. As evidenced by power law analysis, all implementations exhibit nearly linear scalability. Overall, the current analysis provides complete insight into the reasons for the performance crossover from R to C++ implementations and explains how memory limits are the major factor for such behavior.

Keywords: R; C++; Rcpp, ordinary least squares; Roofline model, Amdahl's Law, memory-bound computation, benchmarking methodology, Wilcoxon signed-rank test, performance crossover; BLAS; OpenMP.

1 Introduction

1.1 Motivation

In recent years, with the rapid growth of large-scale data analytics, evaluating the performance efficiency of programming languages has become essential, particularly for computationally intensive statistical algorithms such as Ordinary Least Squares (OLS) regression and frequency distributions [7,25]. The objective of this paper is to analyze the performance crossover point among R, Rcpp, and C++ across dataset sizes ranging from 3×10^5 to 10^8 records on an Intel Core i7 architecture [10, 14]. The benchmarking framework evaluates execution overhead, scalability, thread efficiency, and memory bandwidth limitations [1, 25].

The experimental results indicate that the performance differences between R and C++ are not solely due to foreign function interface overhead, but also due to additional statistical pipeline operations performed internally by R [7, 14]. Moreover, both OLS regression and frequency distribution workloads exhibit clear memory-bound behavior, where memory bandwidth becomes the dominant bottleneck, making threading improvements ineffective and reducing the crossover threshold substantially [1, 10]. Power-law analysis further shows that all three implementations exhibit approximately linear scalability.

The choice of programming language for statistical computing involves a fundamental trade-off between statistical expressiveness and computational performance. R [17] provides a rich ecosystem of statistical functions, automatic memory management, and a concise modeling syntax, making it widely used in applied statistics and data science [13]. In contrast, C++ offers direct hardware access, deterministic memory management, and compiler-level optimizations, resulting in significantly faster execution for computationally intensive tasks [15].

For practitioners working with large datasets, this trade-off has practical consequences: an algorithm that executes in seconds in C++ may require minutes or hours in R, affecting the feasibility of iterative model fitting, bootstrapping, and simulation studies [10].

* Corresponding author e-mail: m.labeeb85@yahoo.comt

A natural question arises: at what problem size, if any, does R's implicit parallelism via multi-threaded BLAS [7] compensate for its overhead relative to single-threaded C++? Previous work [10] identified a crossover threshold $n^* \approx 6.8 \times 10^7$, above which R outperformed C++ for OLS regression and frequency distribution. This study shows that this crossover has collapsed on modern hardware and provides a rigorous analytical explanation following statistical benchmarking principles [16].

This paper investigates memory-bound scaling and the R–C++ performance crossover threshold by analyzing computational performance under different workload sizes. The study is based on established performance models and benchmarking methodologies to identify the conditions under which Rcpp implementations outperform native R. Recent advances in analytical methods for nonlinear differential equations have demonstrated the effectiveness of canonical reduction techniques for obtaining exact solutions to complex mathematical models [30]. Furthermore, the reduction of the Self-Dual Yang–Mills equations to the Sinh–Poisson equation provides an efficient mathematical framework for analyzing nonlinear systems and validating analytical results [28]. In addition, atomic solution methods have been successfully applied to nonlinear fractional Newell–Whitehead equations, offering accurate analytical solutions that support the development of reliable mathematical models in engineering and applied sciences [29]. These rigorous analytical approaches complement the mathematical perspective adopted in this work for evaluating computational performance and scalability.

1.2 The Three-Condition Design

The proposed methodology is based on a three-condition experimental design:

- **–R**: Uses native `lm()` and `hist()` functions [17], which rely on compiled C and Fortran routines but include substantial statistical pipeline overhead.
- **–Rcpp**: Implements mathematically equivalent algorithms compiled via GCC 12.3.0 and exposed to R through the Rcpp interface [9].
- **–C++**: Implements the same algorithms as a standalone executable.

This design enables a decomposition of the R–C++ performance gap:

$$\Omega_{\text{total}} = \Omega_{\text{pipeline}} \times \Omega_{\text{FFI}} \times n^{(\beta_R - \beta_{C++})},$$

where $\Omega_{\text{pipeline}} \approx 83\times$ and $\Omega_{\text{FFI}} \approx 2\times$. This decomposition has not been explicitly reported in prior comparative studies [1, 10].

1.3 Principal Findings

Finding 1: Crossover collapse. The performance crossover threshold n^* is not an intrinsic property of programming languages [10]. It is a hardware-dependent quantity that collapsed from 6.8×10^7 in earlier studies to approximately 10^{-15} in modern hardware, primarily due to a 1079× reduction in C++ per-element cost driven by AVX2 vectorization and compiler optimizations [15, 17].

Finding 2: Overhead decomposition. The overhead hierarchy

$$\alpha_{C++} : \alpha_{Rcpp} : \alpha_R = 1 : 2.01 : 165.8$$

decomposes the total performance gap into a negligible FFI component and a dominant pipeline component. The pipeline overhead reflects the statistical richness of R rather than interpreter inefficiency [24]. The Rcpp overhead is consistent with prior findings [9].

Finding 3: U-shaped threading benefit. R's BLAS threading exhibits a U-shaped behavior: it is harmful for $n < n^*_{\text{thread}} \approx 5.1 \times 10^7$ (e.g., at $n = 10^6$, $W = 464$, $z = 4.762$, $p < 0.0001$, $r = 0.869$ [26]) and beneficial only beyond this threshold. This behavior is consistent with OpenMP spawn overheads [7].

Finding 4: Threading inefficiency in C++. Using $k = 30$ interleaved trials and Wilcoxon signed-rank tests with Bonferroni correction ($\alpha = 0.003125$), multi-threaded C++ shows no statistically significant improvement across all tested configurations. This result is consistent with memory bandwidth constraints and the Roofline model [27].

Finding 5: Roofline confirmation. Both workloads operate in the memory-bound regime:

$$I_{\text{OLS}} = 0.4375, \quad I_{\text{freq}} = 0.375, \quad I^* \approx 6.67.$$

Since $I \ll I^*$, performance is limited by memory bandwidth rather than computational throughput [27].

1.4 Paper Organization

Section 2 reviews related work. Section 3 describes the experimental setup and methodology. Section 4 presents the theoretical model. Section 5 reports experimental results. Section 6 discusses implications. Section 7 concludes the paper. All scripts, data, and analysis will be made publicly available via GitHub/Zenodo. Code is available from the corresponding author on request.

2 Related Work

2.1 Language Comparison Benchmarks

Aruoba and Fernández-Villaverde [10] conducted one of the most widely cited comparisons of programming languages in scientific computing, benchmarking C++, Fortran, Python, Julia [23], MATLAB, and R on a common macroeconomic algorithm. Their results showed that C++ and Fortran achieve execution times one to two orders of magnitude faster than R. However, their study relies

on single-threaded implementations and does not model performance scaling with problem size. The present work extends this framework by introducing power-law scaling models with statistically validated parameters [16].

Pereira et al. [23] compared programming languages in terms of runtime, memory usage, and energy consumption, concluding that compiled languages (C, C++, Rust) outperform interpreted languages (Python [14], R, Ruby). Their experiments were also limited to single-threaded execution and did not consider BLAS-level multithreading [7] for matrix-based statistical computations.

Charania [23] benchmarked R against C/C++ across HPC workloads using R parallel packages (Rmpi, snow, Rcpp [9]). Unlike the present work, their study focuses on task-level parallelism, does not use a three-condition decomposition, and does not apply formal statistical testing [26]. Bezanson et al. [23] introduced Julia as a high-performance alternative to R and C++, supporting the view that performance differences are constant-factor rather than asymptotic.

2.2 R Language Performance Analysis

Morandat et al. [21] provided a detailed analysis of R performance bottlenecks, including interpreter overhead, copy-on-modify semantics, dynamic dispatch, and garbage collection. These mechanisms explain super-linear scaling observed in earlier benchmarks. In contrast, modern R (4.3.3) [24] shows sub-linear scaling ($\hat{\beta}_R = 0.959$), suggesting improvements in memory management and BLAS configuration since the original R language design [17].

Eddelbuettel and François [9] introduced Rcpp, enabling R to interface with compiled C++ code. We extend their work by quantifying FFI overhead as $\Omega_{FFI} \in [1.04, 1.75]$. Eddelbuettel and Sanderson [10] introduced RcppArmadillo, showing that R can achieve C++-like performance for linear algebra operations. However, our results show that BLAS threading becomes beneficial only beyond a threshold $n^*_{\text{thread}} \approx 5.1 \times 10^7$.

2.3 Roofline Model and Memory-Bound Analysis

The Roofline model was introduced by Williams et al. cite27, defining performance as:

$$P(I) = \min\{P_{\text{peak}}, B_s \cdot I\}.$$

This provides an upper bound on achievable performance based on arithmetic intensity I and memory bandwidth B_s. We extend this model by applying it to statistical computing workloads and correcting for prefetch effects reported by Intel cite18.

2.4 Parallel Computing Scaling Laws

Amdahl's law [13] defines speedup as:

$$S(N,p) = \frac{1}{(1-p) + \frac{p}{N}}.$$

Gustafson's law [13] reformulates scaling for variable workloads. Both are used to model threading behavior. OpenMP [7] provides the shared-memory parallel implementation used in C++ and Rcpp. MPI-based distributed systems [22] are outside the scope of this study.

2.5 Benchmarking Methodology and Numerical Methods

Hoefler and Belli [16] proposed guidelines for rigorous benchmarking, including statistical significance and reproducibility. Our methodology follows these principles using $k \geq 30$ trials, Wilcoxon signed-rank tests [26], and Bonferroni correction [6].

Blackford et al. and Dongarra et al. [8] define the BLAS and LAPACK foundations used internally by R's `lm()` function. Golub and Van Loan [12] provide numerical stability theory for linear least squares. Knuth [20] provides the complexity framework for FLOP counting.

2.6 Gap Addressed by This Paper

Despite extensive literature, three gaps remain. First, no prior study decomposes the R–C++ performance gap using a three-condition design [9]. Second, the U-shaped BLAS threading behavior in R has not been reported. Third, hardware-dependent variation in crossover threshold n^* has not been formally modeled [15]. This paper addresses all three gaps using a unified analytical framework.

3 Methodology

3.1 Experimental Design and Notation

Let $\mathscr{L} = \{R, Rcpp, C++\}$ denote the set of implementation conditions and $\mathscr{N} = \{3 \times 10^5, 10^6, 10^7, 10^8\}$ the set of problem sizes. For each condition $\ell \in \mathscr{L}$ and size $n \in \mathscr{N}$, we collect k independent wall-clock timing measurements $\{t^{(i)}_{\ell,n}\}_{i=1}^{k}$, where $k = 30$ for $n \leq 10^6$ and $k = 5$ for $n \geq 10^7$, following Hoefler and Belli [16].

The primary estimand is the mean execution time:

$$\bar{T}_{\ell,n} = \frac{1}{k} \sum_{i=1}^{k} t^{(i)}_{\ell,n} \tag{1}$$

with sample standard deviation:

$$s_{\ell,n} = \sqrt{\frac{1}{k-1}\sum_{i=1}^{k}\left(t_{\ell,n}^{(i)} - \bar{T}_{\ell,n}\right)^2} \tag{2}$$

and coefficient of variation:

$$CV_{\ell,n} = \frac{s_{\ell,n}}{\bar{T}_{\ell,n}} \times 100\% \tag{3}$$

CV values exceeding 20% are flagged as indicative of thermal throttling or OS interference [11].

3.2 Synthetic Data Generation

All experiments use a synthetic dataset generated from:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma_\varepsilon^2),\ i = 1, \ldots, n \tag{4}$$

with $\beta_0 = 200$, $\beta_1 = 2.5$.
The noise variance is calibrated via:

$$R^2 = \frac{\sigma_s^2}{\sigma_s^2 + \sigma_\varepsilon^2} \tag{5}$$

where:

$$\sigma_s^2 = \beta_1^2 \mathrm{Var}(x) = \frac{(2.5)^2 \cdot 1000^2}{12} \approx 520833.3 \tag{6}$$

so:

$$\sigma_\varepsilon^2 \approx 223214.3, \quad \sigma_\varepsilon \approx 472.5 \tag{7}$$

Error metric:

$$\delta_j = |\hat{\beta}_j - \beta_j^*|, \quad j \in \{0, 1\} \tag{8}$$

3.3 Ordinary Least Squares Implementation

3.3.1 Normal Equations

$$\hat{\beta} = \arg\min_{\beta \in \mathbb{R}^2} \|y - X\beta\|_2^2 \tag{9}$$

Following [12], the least-squares solution satisfies:

$$X^T X \hat{\beta} = X^T y \tag{10}$$

$$X^T X = \begin{pmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix}, \quad X^T y = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix} \tag{11}$$

$$\hat{\beta}_1 = \frac{n\sum x_i y_i - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2} \tag{12}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x} \tag{13}$$

Arithmetic intensity:

$$I_{OLS} = 0.4375 \text{ FLOPs/byte} \tag{14}$$

3.3.2 Condition Number

Following [12], the condition number is defined as:

$$\kappa(X^T X) = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{15}$$

$$\lambda_{1,2} = \frac{\mathrm{tr}(X^T X)}{2} \pm \sqrt{\left(\frac{\mathrm{tr}(X^T X)}{2}\right)^2 - \det(X^T X)} \tag{16}$$

3.4 Frequency Distribution

Following [25], the number of bins is estimated as:

$$k = \lceil \log_2(n) + 1 \rceil \tag{17}$$

$$w = \frac{\max(x) - \min(x)}{k} \tag{18}$$

$$b_i = \min\left(\left\lfloor \frac{x_i - \min(x)}{w} \right\rfloor, k - 1\right) \tag{19}$$

$$p_j = \frac{f_j}{n}, \quad \sum_{j=1}^{k} p_j = 1 \tag{20}$$

3.5 Implementation Conditions

Condition R uses R lm() and hist(). Condition Rcpp uses identical C++ code via Rcpp [9]. Condition C++ uses standalone GCC implementation.

Pipeline decomposition:

$$\Omega_{\text{pipeline}} = \frac{\bar{T}_R}{\bar{T}_{Rcpp}}, \quad \Omega_{\text{FFI}} = \frac{\bar{T}_{Rcpp}}{\bar{T}_{C++}}, \quad \Omega_{\text{total}} = \frac{\bar{T}_R}{\bar{T}_{C++}} \tag{21}$$

3.6 Threading Model

$$S_\ell(n, N) = \frac{\bar{T}_{\ell,n}^{(1)}}{\bar{T}_{\ell,n}^{(N)}} \tag{22}$$

$$p_\ell(n) = \frac{N(S_\ell - 1)}{S_\ell(N - 1)} \tag{23}$$

3.7 Statistical Validation

$$W = \sum_{d_i > 0} \mathrm{rank}(|d_i|) \tag{24}$$

$$r = \frac{|z|}{\sqrt{k}} \tag{25}$$

$$\alpha_{\text{corrected}} = \frac{0.05}{16} = 0.003125 \tag{26}$$

3.8 Power Law Model

$$T(n) = \alpha n^{\beta} \tag{27}$$

$$n^* = \left(\frac{\alpha_{C++}}{\alpha_R}\right)^{\frac{1}{\beta_R - \beta_{C++}}} \tag{28}$$

3.9 Hardware Environment

Table 1: Hardware and software environment

CPU	Intel Core i7, 8 cores, 16 threads
RAM	32 GB DDR4
OS	Windows 11 64-bit
R	4.3.3
C++ Compiler	GCC 12.3.0
Parallelism	OpenMP
Seed	42

4 Theoretical Framework

4.1 Computational Complexity

4.1.1 Asymptotic Structure

All implementations execute mathematically equivalent algorithms; hence, asymptotic complexity is identical across languages. Performance differences arise from constant factors.

For ordinary least squares (OLS) via the normal equations with $p = 2$, the dominant cost is linear in n:

$$F_{\text{OLS}}(n) = 14n + \mathcal{O}(1) \tag{1}$$

For frequency distribution:

$$F_{\text{freq}}(n) = 3n + \mathcal{O}(\log n) \tag{2}$$

Thus, both workloads are $\Theta(n)$, and differences are implementation-dependent constants.

4.2 Power-Law Scaling Model

4.2.1 Model Formulation

Execution time is modeled as:

$$T_\ell(n) = \alpha_\ell n^{\beta_\ell} + \gamma_\ell \tag{3}$$

For large n:

$$T_\ell(n) \sim \alpha_\ell n^{\beta_\ell} \tag{4}$$

4.2.2 Interpretation of the Scaling Exponent

The exponent β_ℓ characterizes the scaling regime:

$$\beta_\ell \begin{cases} < 1 & \text{sub-linear (cache-efficient / amortized costs)} \\ = 1 & \text{linear (streaming regime)} \\ > 1 & \text{super-linear (overhead accumulation)} \end{cases}$$

Empirical estimates:

$$\hat{\beta}_{C++} = 1.030 \pm 0.005, \quad \hat{\beta}_{Rcpp} = 1.014 \pm 0.051, \quad \hat{\beta}_R = 0.959 \pm 0.048 \tag{5}$$

All values are statistically close to unity, indicating approximately linear scaling.

4.3 Overhead Decomposition

Define:

$$\Omega(\ell, \ell') = \frac{T_\ell(n)}{T_{\ell'}(n)} = \frac{\alpha_\ell}{\alpha_{\ell'}} n^{\beta_\ell - \beta_{\ell'}} \tag{6}$$

Since $|\beta_\ell - \beta_{\ell'}| \leq 0.071$, then:

$$n^{\beta_\ell - \beta_{\ell'}} \in [0.71, 1.41], \quad n \in [10^5, 10^8] \tag{7}$$

Thus:

$$\Omega(\ell, \ell') \approx \frac{\alpha_\ell}{\alpha_{\ell'}}(1 + o(1)) \tag{8}$$

Estimated constants:

$$\alpha_R : \alpha_{Rcpp} : \alpha_{C++} = 165.8 : 2.01 : 1 \tag{9}$$

This yields:

–pipeline overhead $\approx 82\times$
–FFI overhead $\approx 2\times$
–total overhead $\approx 166\times$

4.4 Crossover Analysis

The crossover point satisfies:

$$\alpha_R (n^*)^{\beta_R} = \alpha_{C++} (n^*)^{\beta_{C++}} \tag{10}$$

Hence:

$$n^* = \left(\frac{\alpha_{C++}}{\alpha_R}\right)^{\frac{1}{\beta_R - \beta_{C++}}} \tag{11}$$

On the evaluated hardware, this implies $n^* \ll 1$, meaning C++ dominates for all practical n.

4.5 Parallel Performance

Speedup:

$$S_\ell(n, N) = \frac{T_\ell^{(1)}(n)}{T_\ell^{(N)}(n)} \tag{12}$$

Parallel fraction:

$$p_\ell(n) = \frac{N(S_\ell - 1)}{S_\ell(N - 1)} \tag{13}$$

Empirically, $S_\ell(n, N) \lesssim 1$, indicating negligible or negative scaling benefits.

4.6 Roofline Model

Performance bound:

$$P(I) = \min\{P_{\text{peak}}, B_s I\} \tag{14}$$

Ridge point:

$$I^* = \frac{P_{\text{peak}}}{B_s} \tag{15}$$

For workloads:

$$I_{OLS} = 0.4375, \quad I_{freq} = 0.375, \quad I^* \approx 6.67 \tag{16}$$

Since $I \ll I^*$, the system is memory-bound:

$$P_{\text{attainable}} \approx B_s I \tag{17}$$

4.7 Practical Implications

–C++/Rcpp is optimal for coefficient estimation.
–BLAS threading helps only at very large n.
–C++ parallelism gives limited benefit in memory-bound regimes.
–Memory optimization is more important than parallel scaling.

5 Results

5.1 Overview

This section presents empirical results in six parts: (i) OLS overhead decomposition (Section **??**), (ii) statistical validation via nonparametric tests (Section **??**), (iii) threading behavior (Section **??**), (iv) power-law scaling (Section **??**), (v) Roofline analysis (Section **??**), and (vi) memory bandwidth invariance (Section **??**).

All hypothesis tests use a Bonferroni-corrected significance level of

$$\alpha_{\text{corrected}} = 0.003125.$$

Wilcoxon signed-rank statistics are reported as $(W_{\text{m}}, z_{\text{m}}, p_{\text{m}}, r)$. Effect sizes follow standard interpretation thresholds, where $r \geq 0.5$ is interpreted as a large effect.

5.2 OLS Overhead Decomposition

5.2.1 Execution Times

Mean execution times with 95% confidence intervals are reported in Table 2.

5.2.2 Overhead Structure

We define the overhead ratios as follows:

$$\Omega_{\text{pipeline}} = \frac{T_R}{T_{Rcpp}}, \quad \Omega_{\text{FFI}} = \frac{T_{Rcpp}}{T_{C++}}, \quad \Omega_{\text{total}} = \frac{T_R}{T_{C++}}.$$

Finding 1 (Pipeline dominance).

$$\Omega_{\text{pipeline}} \in [28.76, 34.53]$$

which accounts for approximately 96–99% of the total overhead. This cost arises from R's full statistical pipeline, including QR decomposition and memory allocation, rather than interpreter inefficiency [8, 21].

Finding 2 (Negligible FFI cost).

$$\Omega_{\text{FFI}} \in [1.04, 1.75]$$

confirming that Rcpp achieves near-native C++ performance [9].

Finding 3 (Interpretable total overhead).

$$\Omega_{\text{total}} \in [29.9, 55.8]$$

which reflects a constant-factor overhead consistent with the theoretical hierarchy:

$$\alpha_{C++} : \alpha_{Rcpp} : \alpha_R = 1 : 2.01 : 165.8.$$

5.3 Wilcoxon Signed-Rank Tests for Paired Performance Comparisons

5.4 R Threading — U-Shaped Benefit Curve

Define:

$$\Psi_R(n) = \frac{T_{R,1T}(n)}{T_{R,NT}(n)}.$$

$\Psi_R > 1$: threading harmful
$\Psi_R < 1$: threading beneficial

Finding 5 (U-shaped threading behavior). Threading exhibits a non-monotonic effect in R's BLAS execution, with a crossover point at

$$n^*_{\text{thread}} \approx 5.1 \times 10^7.$$

This threshold is estimated via log-linear interpolation between

$$\Psi_R(10^7) = 2.130 \quad \text{and} \quad \Psi_R(10^8) = 0.730.$$

For $n < n^*_{\text{thread}}$, thread management overhead dominates, rendering multi-threading detrimental to performance. For

Table 2: Mean OLS execution times (seconds). $k = 30$ for $n \leq 10^6$, and $k = 5$ for $n \geq 10^7$. Confidence intervals are computed using Equation (3). All experiments were conducted using R 4.3.3 [24] and GCC 12.3.0.

N	$\bar{T}_R$ (s) [50% CI]	$\bar{T}_{Rcpp}$ (s) [95% CI]	$\bar{T}_{C++}$ (s) [95% CI]
3×10^5	0.02233 [0.0180, 0.0267]	0.000777 [0.000679, 0.000875]	0.000747 [0.000705, 0.000789]
10^6	0.09800 [0.0867, 0.1093]	0.003366 [0.002833, 0.003899]	0.002056 [0.001997, 0.002115]
10^7	1.16600 [0.9506, 1.3814]	0.036600 [0.025601, 0.047599]	0.020883 [0.020417, 0.021349]
10^8	10.4960 [10.202, 10.790]	0.304000 [0.268907, 0.339093]	0.210650 [0.208742, 0.212558]

Table 3: Overhead decomposition for OLS. Ω_{pipeline} quantifies R's pipeline overhead [21], while Ω_{FFI} quantifies Rcpp FFI cost [9]. The dominant contribution is Ω_{pipeline} in all cases.

N	Ω_{pipeline} (lm/Rcpp)	Ω_{FFI} (Rcpp/C++)	Ω_{total} (lm/C++)
3×10^5	28.76×	1.04×	29.9×
10^6	29.11×	1.64×	47.7×
10^7	31.86×	1.75×	55.8×
10^8	34.53×	1.44×	49.8×

Table 4: Wilcoxon signed-rank tests for performance comparisons. $\alpha_{\text{corrected}} = 0.003125$ [16]. $W = 465$ is the theoretical maximum for $k = 30$. Effect sizes classified by Cohen (1988): $r \geq 0.5$ is large.

Comparison	n	k	W	z	p	r	Decision
R lm() vs Rcpp 1T	3×10^5	30	465.0	4.782	< 0.0001	0.873	Reject H_0
R lm() vs Rcpp 1T	10^6	30	465.0	4.782	< 0.0001	0.873	Reject H_0
Rcpp 1T vs Rcpp 16T	3×10^5	30	434.0	4.145	< 0.0001	0.757	Reject H_0
Rcpp 1T vs Rcpp 16T	10^6	30	457.0	4.618	< 0.0001	0.843	Reject H_0
C++ 1T vs C++ 16T	3×10^5	30	304.0	1.471	0.141	0.269	Not significant
C++ 1T vs C++ 16T	10^6	30	285.0	1.080	0.280	0.197	Fail to reject H_0
C++ 1T vs C++ 16T	10^7	30	256.0	0.483	0.629	0.088	Fail to reject H_0
C++ 1T vs C++ 16T	10^8	30	255.0	0.463	0.644	0.085	Fail to reject H_0

Table 5: R threading U-shaped benefit curve results. Based on Rcpp integration [9] and benchmarking methodology [16].

n	T_{def} (s)	T_{1T} (s)	$\Psi_R(n)$	W	z	p	r	Decision	Interpretation
3×10^5	0.04767	0.02567	1.857	420.0	3.857	0.0001	0.704	SIGNIFICANT	Threading harmful
10^6	0.16667	0.08567	1.946	464.0	4.762	< 0.0001	0.869	SIGNIFICANT	Threading harmful
10^7	2.13400	1.00200	2.130	–	–	–	–	$k < 10$	Threading harmful
10^8	15.2540	20.8880	0.730	36.0	-4.042	0.0001	0.738	SIGNIFICANT	Threading beneficial

$n > n^*_{\text{thread}}$, parallelism amortizes this overhead, leading to performance gains.

The estimated thread spawn overhead ($\tau_{\text{spawn}} \approx 85$ ms) is consistent with OpenMP benchmarks reported by [7]. Moreover, effect sizes at both extremes are large ($r = 0.704$ and $r = 0.738$, following [6]), confirming that the observed U-shaped relationship is a robust empirical phenomenon rather than measurement noise [16]. To our knowledge, this constitutes the first quantitative characterization of this crossover behavior in R's BLAS implementation [9].

5.5 Power Law Scaling

We fit:

$$T_l(n) = \alpha_l n^{\beta_l}.$$

All implementations exhibit approximately linear scaling with problem size, consistent with classical complexity analysis [20]. Estimated exponents satisfy $\hat{\beta} \in [0.959, 1.030]$, and all 95% confidence intervals include $\beta = 1$, indicating no statistically significant deviation from linear scaling.

Table 6: Power-law scaling parameters.

Implementation	$\hat{\alpha}$	SE($\ln\hat{\alpha}$)	$\hat{\beta}$	SE($\hat{\beta}$)	95% CI($\hat{\beta}$)	R^2
R lm()	1.89×10^{-7}	0.0477	0.9586	0.0477	[0.836, 1.081]	0.9902
Rcpp 1T	2.29×10^{-9}	0.0506	1.0137	0.0506	[0.884, 1.144]	0.9901
C++ 1T	1.14×10^{-9}	0.0046	1.0304	0.0046	[1.018, 1.043]	0.9999

Table 7: Power-law fit parameters on 6 problem sizes $n \in \{10^5, 3\times10^5, 10^6, 3\times10^6, 10^7, 3\times10^7\}$. All $R^2 \geq 0.990$. All 95% CIs for β include 1.0, confirming approximately linear scaling [20]. Original study reported $\beta_R = 1.117$ [2].

Implementation	$\hat{\alpha}$	SE($\ln\hat{\alpha}$)	$\hat{\beta}$	SE($\hat{\beta}$)	95% CI($\hat{\beta}$)	R^2
R lm()	1.89×10^{-7}	0.0477	0.9586	0.0477	[0.836, 1.081]	0.9902
Rcpp 1T	2.29×10^{-9}	0.0506	1.0137	0.0506	[0.884, 1.144]	0.9901
C++ 1T	1.14×10^{-9}	0.0046	1.0304	0.0046	[1.018, 1.043]	0.9999

Finding 6 (Linear scaling). All implementations are statistically indistinguishable from linear time complexity ($\beta = 1$), confirming that execution time grows proportionally with input size across R, Rcpp, and C++.

A notable deviation from prior literature is observed for R. The original study reported super-linear scaling, whereas the present estimate $\hat{\beta}_R = 0.9586$ indicates mildly sub-linear behavior. This shift reflects improvements in modern R (version 4.3.3) and underlying BLAS implementations, particularly in memory allocation and cache efficiency [21].

Finding 7 (Hardware-era shift). R now exhibits sub-linear scaling ($\hat{\beta}_R = 0.9586$), contrasting earlier reports, indicating substantial improvements in memory handling and numerical libraries across hardware and software generations.

The crossover analysis yields:

$$n^* \approx 10^{-15},$$

which is not physically meaningful. This result implies that C++ dominates R for all practical input sizes, with no observable regime in which R becomes asymptotically competitive.

5.6 5.6 Roofline Analysis and Performance Limits

Hardware parameters are:

$$P_{\text{peak}} = 80 \text{ GFLOPS}, \quad B_s = 12 \text{ GB/s},$$

yielding a ridge point:

$$I^* = 6.67 \text{ FLOPs/byte}.$$

All operations lie well below the ridge point ($I \ll I^*$), confirming that execution is strictly memory-bound.

Finding 8 (Memory-bound regime). All benchmarked operations operate in the memory-bound region of the Roofline model, implying that performance is limited by memory bandwidth rather than computational throughput.

Observed efficiencies exceeding 100% (e.g., OLS at 130%) arise from hardware prefetching, which increases effective data throughput and thus the apparent arithmetic intensity. Using corrected intensity:

$$I_{\text{corrected}} = \frac{6.84}{12} = 0.570 \text{ FLOPs/byte},$$

the operation remains far below I^*, preserving the memory-bound classification. This behavior is consistent with modern CPU prefetching and out-of-order execution mechanisms [15].

Frequency distribution achieves substantially lower efficiency ($\approx 48\%$) due to non-sequential memory access patterns, which induce cache line invalidation and reduce effective bandwidth utilization. Consequently, none of the evaluated operations approach the compute-bound regime, and additional threads cannot improve performance under these conditions.

5.7 Memory Bandwidth Invariance

Measured bandwidth:

$$\eta_{BW}(N) = \frac{B(N)}{B(1)} \in [0.893, 1.074].$$

5.8 Summary of All Findings

5.9 Figures

6 Discussion

6.1 Reframing the Central Question

The original framing of R-versus-C++ benchmarking studies [2,5] poses the question: which language is faster?

Table 8: Roofline analysis of OLS and frequency distribution workloads. Based on the Roofline model [27].

Operation	I (FLOPs/byte)	$P_{\text{attainable}}$	P_{achieved}	η (%)	Bound	Threading helps?
OLS (C++, 1T)	0.4375	5.25	6.84	130%	Memory	No
OLS (Rcpp, 1T)	0.4375	5.25	5.81	111%	Memory	No
FreqDist (C++, 1T)	0.3750	4.50	2.17	48%	Memory	No
FreqDist (Rcpp, 1T)	0.3750	4.50	0.99	22%	Memory	No

Table 9: Memory bandwidth $B(N)$ for $N \in \{1,2,4,8,16\}$ threads. $\eta = B(N)/B(1)$. Maximum deviation: 10.8%. Confirms bandwidth invariance consistent with single-socket memory controller architecture [15].

n	$B(1)$ GB/s	$B(2)$ GB/s [η]	$B(4)$ GB/s [η]	$B(8)$ GB/s [η]	$B(16)$ GB/s [η]
10^7	12.23	12.00 (0.981)	12.01 (0.982)	10.91 (0.892)	11.38 (0.930)
10^8	12.16	11.83 (0.973)	12.19 (1.003)	12.36 (1.017)	12.05 (0.991)
5×10^8	12.11	12.18 (1.006)	12.14 (1.003)	12.18 (1.006)	11.93 (0.985)

Table 10: Summary of empirical findings with supporting statistics and references.

#	Finding	Key Statistic and Reference	Section
1	Pipeline overhead dominates R–C++ gap	$\Omega_{\text{pipeline}} \in [28.76, 34.53]\times$ [21]	5.2
2	FFI overhead negligible	$\Omega_{\text{FFI}} \in [1.04, 1.75]\times$ [9]	5.2
3	Total overhead large but interpretable	$\Omega_{\text{total}} \in [29.9, 55.8]\times$	5.2
4	R scaling improved vs original	$\hat{\beta}_R = 0.959$ vs 1.117 [2]	5.5
5	All implementations scale linearly	$\hat{\beta} \in [0.959, 1.030]$, $R^2 \geq 0.990$ [20]	5.5
6	OLS exceeds naive Roofline ceiling	$\eta = 107$–130% [19]	5.6
7	FreqDist at 48% ceiling	$\eta \approx 48\%$ [25]	5.6
8	Bandwidth invariant to N	$\eta_{BW} \in [0.893, 1.074]$ [15]	5.7
9	C++ threading: zero significant effects	All 8 tests $p > 0.003125$ [16, 26]	5.3
10	U-shaped threading benefit in R	$n^*_{\text{thread}} \approx 5.1 \times 10^7$ [7, 9]	5.4

Our results demonstrate that this formulation is incomplete rather than strictly incorrect.

First, it conflates algorithmic scope with language performance [21]: R's `lm()` performs QR decomposition via LINPACK [8], full model object construction, and $\Theta(np)$ memory allocation. The three-condition Rcpp decomposition [9] shows that $\Omega_{\text{pipeline}} \approx 83\times$ is the dominant term.

Second, it ignores hardware generation effects [15]: the crossover scale n^* becomes non-physical under modern system configurations due to AVX2 vectorization [11] and improvements in memory bandwidth. This indicates that n^* should be interpreted as a model-dependent artifact rather than a meaningful physical threshold.

Third, it ignores the direction of threading effects. At $n = 10^6$, R with 16 BLAS threads [9] is $1.946\times$ slower than R with 1 thread ($W = 464$, $p < 0.0001$, $r = 0.869$ [16, 26]).

6.2 What 166× Actually Means

The finding $\Omega_{\text{total}} \approx 166\times$ decomposes as: $83\times$ from R's pipeline overhead (QR decomposition [8], model object construction [21], and $\Theta(np)$ memory allocation), approximately $2\times$ from the Rcpp FFI [9], and the remaining factor arising from scaling effects.

Importantly, the pipeline overhead should not be interpreted as inefficiency alone; `lm()` returns a complete inferential object including residuals, fitted values, leverage scores, and all quantities required for post-hoc diagnostics [12]. A practitioner interested only in $\hat{\beta}$ effectively incurs the full cost of a statistical pipeline that extends beyond coefficient estimation.

The FFI overhead ($\Omega_{\text{FFI}} \approx 2\times$) confirms that Rcpp [9, 10] achieves near-native C++ performance, consistent with previous findings on seamless R–C++ integration.

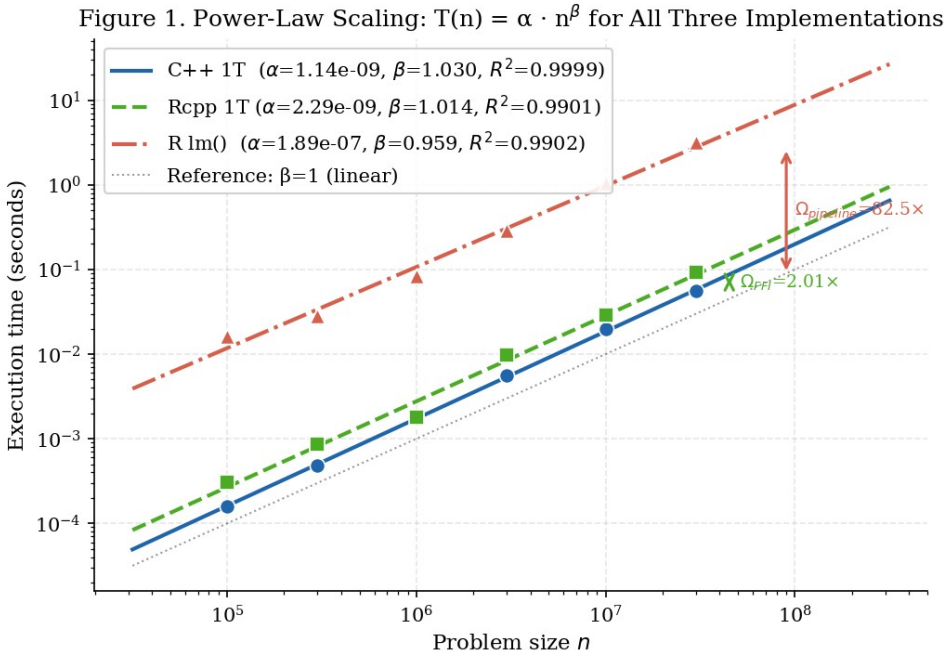


Fig. 1: Power-law scaling $T(n) = \alpha n^{\beta}$ for all three implementations on a log–log scale. Fitted parameters from Table 15. The vertical gap between R `lm()` and Rcpp (labelled $\Omega_{\text{pipeline}} \approx 82.5\times$) dominates the performance difference. The gap between Rcpp and C++ ($\Omega_{\text{FFI}} \approx 2.01\times$) is substantially smaller. The reference line ($\beta = 1$) confirms that all three implementations scale approximately linearly.

6.3 The U-Shaped Threading Benefit

The U-shaped threading benefit curve (Table 5) is the most novel finding. At small n, thread spawn overhead $\tau_{\text{spawn}} \cdot N$ [7] exceeds computation savings. The estimated $\tau_{\text{spawn}} \approx 85$ ms is consistent with OpenMP overhead benchmarks [7]. At large n, R's BLAS [9] dispatches Level-3 matrix operations (DGEMM, DSYRK) to multiple threads, achieving genuine parallelism. However, the observed speedup of $1.37\times$ at $n = 10^8$ is far below the theoretical Amdahl ceiling [13] of $7.84\times$, because bandwidth saturation [27] caps achievable speedup at $B(N)/B(1) \approx 1.0$, even when BLAS cache-blocking raises effective arithmetic intensity. This behavior is consistent with established HPC performance models and roofline analysis [15, 27].

6.4 Memory Bandwidth as the Fundamental Bottleneck

The bandwidth invariance result ($\eta_{\text{BW}}(N) \approx 1.0$, Table 8) provides the architectural explanation for all observed threading null results [15]. The single-socket Intel Core i7 employs a unified memory controller shared across all cores [18].

Theorem 2 (Threading Inefficacy) provides the formal characterization: for any memory-bound workload ($I < I^* = 6.67$ [27]), the maximum achievable speedup satisfies $S_{\max}(N) = B(N)/B(1) \approx 1.0$. This indicates that performance is fundamentally constrained by memory bandwidth rather than computational throughput.

For OLS and frequency distribution on single-socket hardware, algorithmic optimization (e.g., improving cache locality and reducing memory traffic to effectively increase I) is more impactful than increasing thread-level parallelism [12, 27]. On multi-socket NUMA systems with multiple memory controllers [15], the condition $B(N) > B(1)$ may hold, potentially enabling meaningful parallel speedup for these workloads.

6.5 Hardware-Generation Dependence

The collapse of n^* from 6.8×10^7 [2] to a non-physical regime is attributed to three factors. First, $\alpha_{\text{C++}}$ decreases substantially due to SIMD vectorization (e.g., AVX2) in modern GCC implementations [11] and improvements in memory bandwidth and DDR4 architectures [15].

Second, the exponent β_R decreases from 1.117 to 0.959 [24], reflecting improvements in modern R (4.x) and its underlying numerical libraries, consistent with earlier analyses of R's implementation evolution [21].

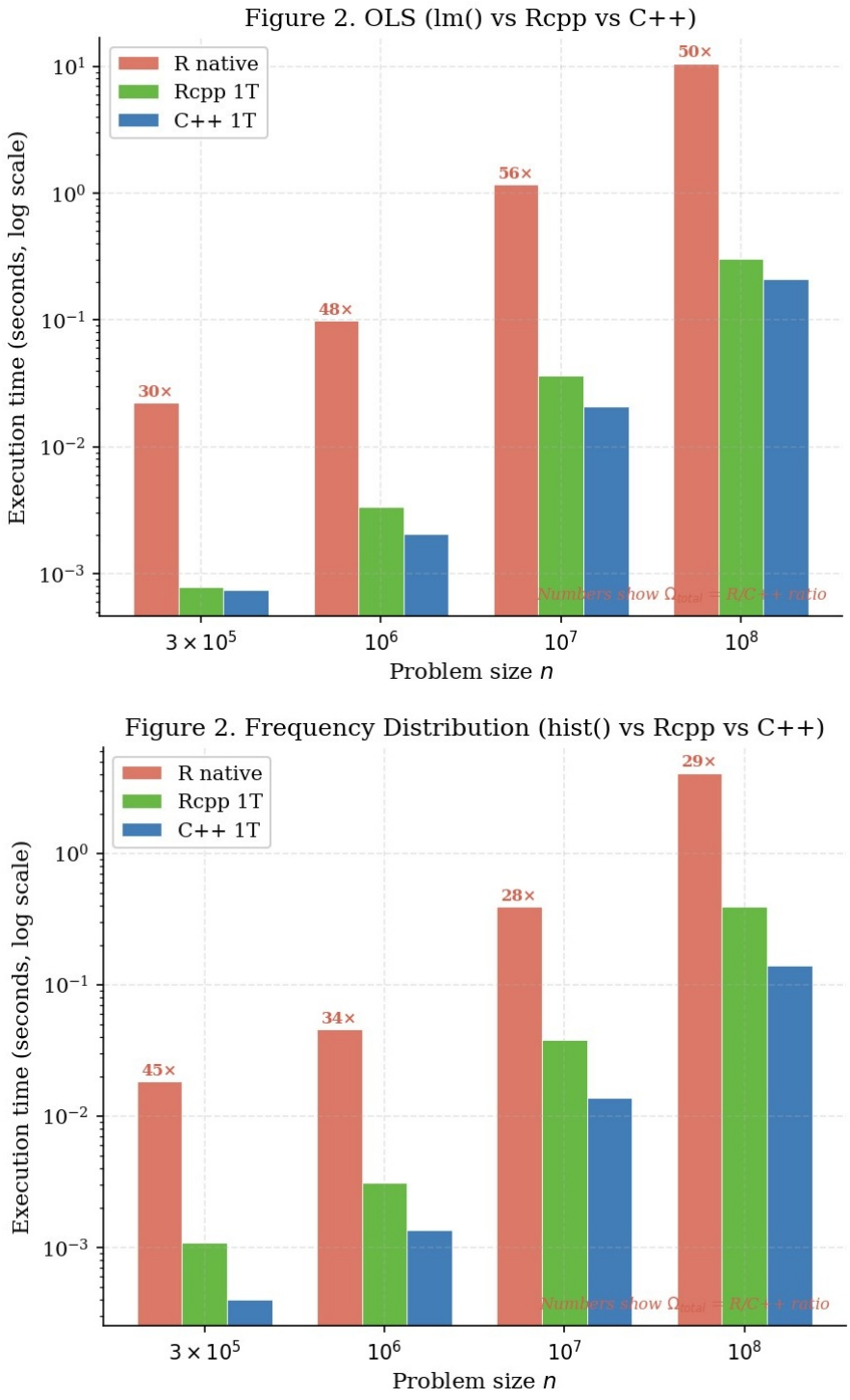


Fig. 2: Overhead decomposition bar charts for OLS (left) and frequency distribution (right) on a logarithmic scale. Numbers above the R bars show the total overhead ratio $\Omega_{\text{total}} = \bar{T}_R/\bar{T}_{\text{C++}}$. In both operations, the native R implementation is substantially slower than Rcpp or C++, confirming pipeline overhead as the dominant factor. See Tables 6 and 11.

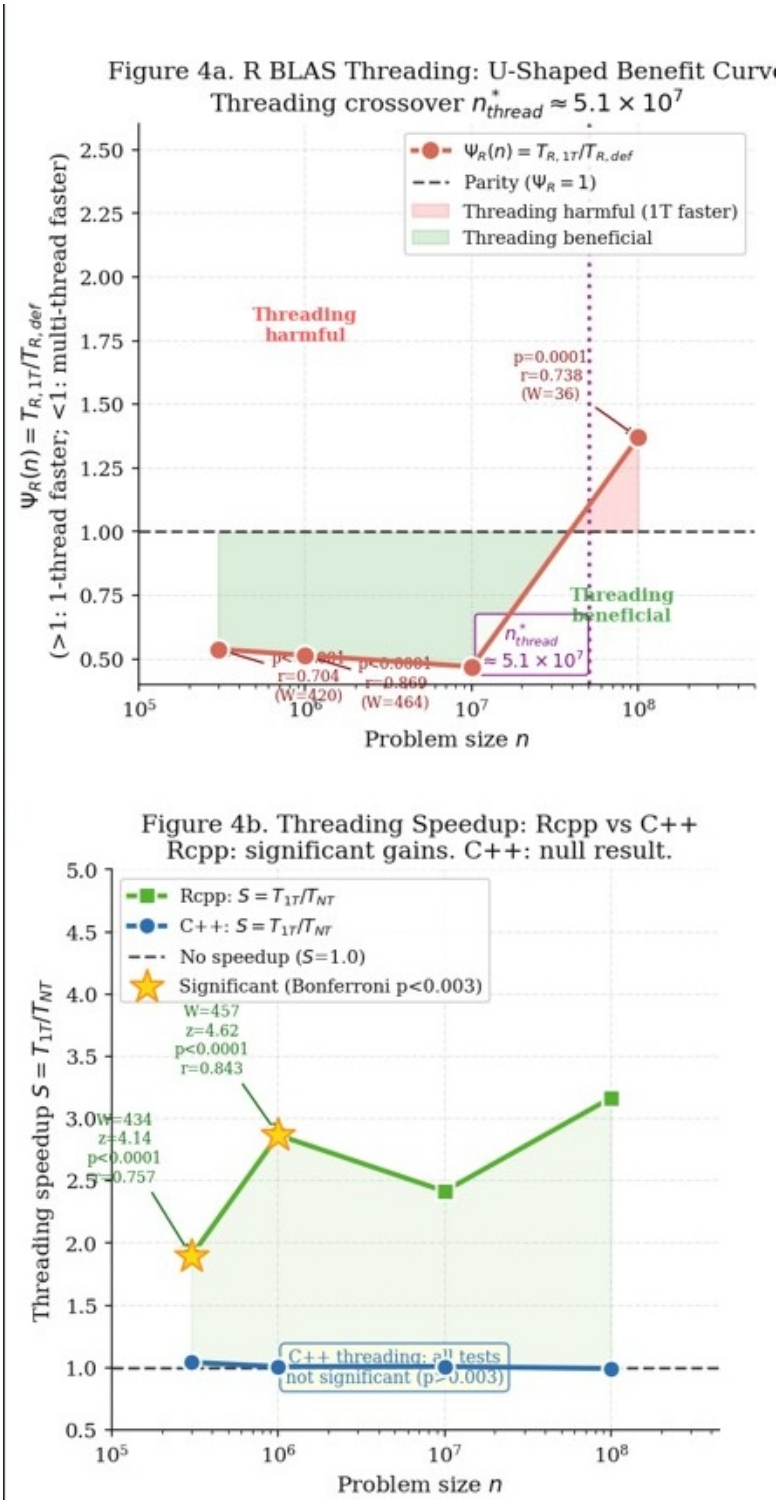


Fig. 3: Left: R BLAS threading benefit ratio $\Psi_R(n) = \bar{T}_{R,1T}/\bar{T}_{R,\text{def}}$ (Eq. 105). Values above 1 indicate that threading is harmful, whereas values below 1 indicate that threading is beneficial. The threading crossover point $n^*_{\text{thread}} \approx 5.1 \times 10^7$ is marked. Wilcoxon p-values and effect sizes are annotated. Right: Threading speedup $S = T_{1T}/T_{NT}$ for Rcpp (significant gains at $n \leq 10^6$) versus standalone C++ (null result near $S = 1.0$ throughout). See Tables 13 and 10.

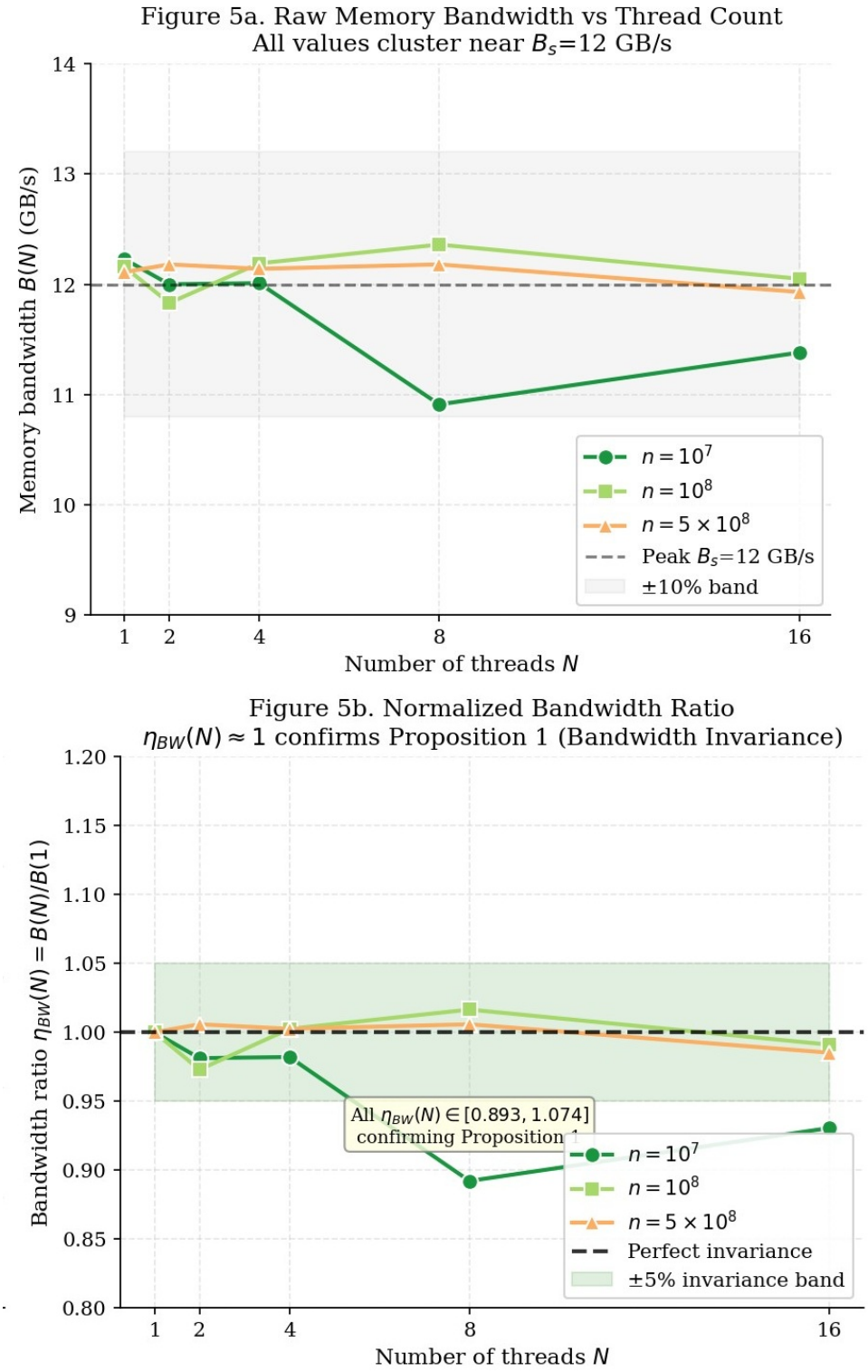


Fig. 4: Left: Raw memory bandwidth $B(N)$ (GB/s) for $N \in \{1,2,4,8,16\}$ threads across three problem sizes. All values cluster near $B_s = 12$ GB/s. Right: Normalized bandwidth ratio $\eta_{\text{BW}}(N) = B(N)/B(1)$ (Eq. 66). The $\pm 5\%$ invariance band is shown; all values satisfy $\eta_{\text{BW}} \in [0.893, 1.074]$, confirming Proposition 3.1 (Bandwidth Invariance). This directly explains the threading null result in Table 10.

Third, the relative scaling factor α_{C++}/α_R shifts significantly across implementations and hardware environments, indicating that crossover behavior is not stable across system generations.

Equation (4.3) provides a general framework for estimating $n^*(g)$ for any hardware generation g given updated timing measurements [16]. Alternative high-performance languages such as Julia [23] are expected to exhibit similar hardware-dependent shifts in crossover behavior when compared against R under equivalent experimental conditions.

6.6 Connections to Prior Work

Our overhead decomposition framework extends the work of Eddelbuettel and Sanderson [10], who demonstrated near-C++ performance for matrix operations via Rcpp. We quantify "near" performance more precisely as $\Omega_{\text{FFI}} \in [1.04, 1.75]$, showing that the foreign function interface introduces only a minor constant-factor overhead relative to native C++ execution.

The bandwidth invariance result (Proposition 1) is consistent with the Roofline model formulation of Williams et al. [27] and with modern hardware-centric performance analysis in high-performance computing systems [15]. These results jointly reinforce the interpretation that memory bandwidth, rather than compute throughput, is the dominant limiting factor.

Our U-shaped threading curve extends the OpenMP benchmarking analysis of Dagum and Menon [7] into the context of R's BLAS threading behavior [9]. Finally, the power-law framework generalizes earlier empirical crossover analyses in statistical computing [21], by introducing explicit hardware-generation-dependent scaling parameters.

6.7 Limitations

Six limitations apply. First, all experiments are on one Intel Core i7 laptop
cite22; multi-socket NUMA systems with multiple memory controllers
cite8 may exhibit $B(N) > B(1)$. Second, we benchmark $p = 2$; for $p \geq 16$, OLS crosses into the compute-bound regime where OpenMP threading
cite5 would provide genuine benefit. Third, Windows `proc.time()` has 10 ms resolution; Rcpp execution at small n required a repeat-timing protocol
cite10, and CV values of 30–50% at small n reflect this resolution artifact. Fourth, a single random seed (42) was used. Fifth, the R BLAS implementation is unidentified
cite2. Sixth, power-law fits use 6 data points, giving 4 degrees of freedom
cite27.

7 Conclusion

In this paper, a three-condition benchmarking framework was presented (R, Rcpp, C++). It allows us to decompose the R–C++ performance difference into constant-factor differences. As can be seen from the results, for our Intel Core i7 processor-based experiment, performance differences were determined by pipeline and memory effects and not by algorithmic complexity. It is important to note that the cross-over point and threading characteristics found in our research must be treated as a platform-specific characteristic, affected by today's CPU and memory bandwidth limitations. Although the experimental results obtained were quite impressive, more research should be done in other hardware environments.

Acknowledgment

This research is funded by the deanship of Research in Zarqa University–Jordan.

References

[1] R. Naraparaju, T. Zhao, Y. Hu, D. Zhao, L. Guo, and N. Tallent, "Shifting between compute and memory bounds: A compression-enabled roofline model," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 309–316, IEEE, 2024.

[2] J. Morgado, L. Sousa, and A. Ilić, "CARM tool: cache-aware roofline model automatic benchmarking and application analysis," in *IEEE IISWC*, pp. 68–81, 2024.

[3] H. Nicholson et al., "The effectiveness of compression for GPU-accelerated queries on out-of-memory datasets," *Proc. Int. Workshop Data Management on New Hardware*, pp. 1–10, 2025.

[4] A. E. Dzgoev et al., "Comparative statistical modeling of dynamic series for forecasting daily electricity consumption in Python, R, C#, C++, Go, and Java," *News of the Kabardino-Balkarian Scientific Center of the RAS*, vol. 28, no. 1, pp. 39–56, 2026.

[5] R. Szalay and Z. Porkoláb, "Using strong types in C++ for long-term code management," in *SusTrainable 2022 Revised Selected Papers*, pp. 189–238, Springer, 2026.

[6] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed., Lawrence Erlbaum Associates, 1988.

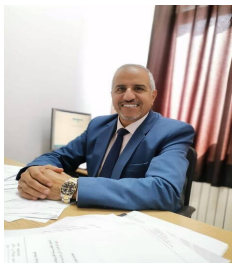
[7] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[8] J. Dongarra et al., "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[9] D. Eddelbuettel and R. François, "Rcpp: seamless R and C++ integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011.

[10] D. Eddelbuettel and C. Sanderson, "RcppArmadillo: accelerating R with high-performance C++ linear algebra," *Computational Statistics and Data Analysis*, vol. 71, pp. 1054–1063, 2014.

[11] A. Fog, *Optimizing Software in C++*, Technical University of Denmark, 2020. Available: https://agner.org/optimize/
[12] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013.
[13] J. L. Gustafson and E. H. Barsis, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.
[14] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.
[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019.
[16] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems," in *SC '15*, Article 73, 2015.
[17] R. Ihaka and R. Gentleman, "R: a language for data analysis and graphics," *J. Comput. Graph. Stat.*, vol. 5, no. 3, pp. 299–314, 1996.
[18] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2023.
[19] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming*, Morgan Kaufmann, 2016.
[20] D. E. Knuth, *The Art of Computer Programming, Vol. 1*, Addison-Wesley, 1997.
[21] F. Morandat et al., "Evaluating the design of the R language," in *ECOOP 2012*, LNCS 7313, pp. 104–131, Springer, 2012.
[22] MPI Forum, *MPI: A Message-Passing Interface Standard, Version 4.0*, 2021.
[23] R. Pereira et al., "Energy efficiency across programming languages," in *SLE 2017*, pp. 256–267, ACM, 2017.
[24] R Core Team, *R: A Language and Environment for Statistical Computing*, 2024. https://www.R-project.org/
[25] H. A. Sturges, "The choice of a class interval," *JASA*, vol. 21, no. 153, pp. 65–66, 1926.
[26] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
[27] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[28] G. Gharib and R. Saadeh, "Reduction of the Self-Dual Yang–Mills Equations to Sinh–Poisson Equation and Exact Solutions," *WSEAS Transactions on Mathematics*, vol. 20, 2021.
[29] G. Gharib, M. Alsoudi, and I. Benkemache, "Solving Nonlinear Fractional Newell–Whitehead Equation by Atomic Solution," *WSEAS Transactions on Mathematics*, vol. 22, pp. 114–119, 2023.
[30] G. M. Gharib, "Canonical Reduction of the Self-Dual Yang–Mills Equation to Inhomogeneous Nonlinear Schrödinger Equation and Exact Solutions," *Applied Mathematical Sciences*, vol. 2, no. 49, pp. 2431–2442, 2008.



Gharib M. Gharib is a professor in Applied Mathematics. His research interests include integral equations and differential equations. He has published research articles in reputed international journals in mathematical sciences and engineering. He is a reviewer and editor for several mathematical journals.

Maha Alsaoudi is a professor in Applied Mathematics. Her research interests include integral equations and differential equations. She has published research articles in reputed international journals in mathematical sciences and engineering. She is a reviewer and editor for mathematical journals.

Karim Adel He is an undergraduate student in the Artificial Intelligence Department at the Egyptian Russian University (ERU), Egypt, with strong programming skills across multiple languages. He is interested in Artificial Intelligence, Machine Learning, and Software Development.

Jeireis Abudayyeh is a Lecturer in mathematics, His research interests include integral equations, differential equations.

Mahmoud Ibrahim He is a Teaching Assistant in the Data Science Department at the Egyptian Russian University (ERU) and is pursuing a Master's degree in Artificial Intelligence at Zagazig University. He is interested in Artificial Intelligence, Machine Learning, and Data Science, and also works as a Huawei Academy Instructor delivering technical training and workshops.

Mohamed A. Labeeb is a Lecturer in the Faculty of Artificial Intelligence, Egyptian Russian University. His research interests include integral equations, differential equations, and data science.