Benchmark Report: `linear-massiv` vs. `hmatrix` vs. `linear`

Performance Comparison of Haskell Linear Algebra Libraries

Nadia Chambers   Claude Opus 4.6

February 2026

Abstract

We present a comprehensive performance comparison of three Haskell numerical linear algebra libraries: `linear-massiv` (pure Haskell, type-safe dimensions via massiv arrays), `hmatrix` (FFI bindings to BLAS/LAPACK via OpenBLAS), and `linear` (pure Haskell, optimised for small fixed-size vectors and matrices). Benchmarks cover BLAS-level operations, direct solvers, orthogonal factorisations, eigenvalue problems, and singular value decomposition across matrix dimensions from 4×4 to 200×200. Additionally, we evaluate the parallel scalability of `linear-massiv`'s massiv-backed computation strategies on a 20-core workstation. Results show that `hmatrix` (OpenBLAS) dominates at all sizes for $O(n^3)$ operations due to highly-optimised Fortran BLAS/LAPACK routines, while `linear` excels at 4×4 through unboxed product types. `linear-massiv` provides competitive pure-Haskell performance with the unique advantages of compile-time dimensional safety, no FFI dependency, and user-controllable parallelism that yields 3–7× speedups at larger matrix sizes.

Contents

1 Introduction

The Haskell ecosystem offers several numerical linear algebra libraries, each occupying a distinct niche:

`linear` Edward Kmett's library provides small fixed-dimension types (`V2`, `V3`, `V4`) with unboxed product representations, making it extremely fast for graphics, game physics, and any application where dimensions are statically known and small. It does not support arbitrary-dimension matrices.

`hmatrix` Alberto Ruiz's library wraps BLAS and LAPACK via Haskell's FFI, delegating numerical computation to highly-optimised Fortran routines (on this system, OpenBLAS). It supports arbitrary dimensions but carries an FFI dependency and provides no compile-time dimension checking.

`linear-massiv` Our library implements algorithms from Golub & Van Loan's *Matrix Computations* (4th ed.) [1] in pure Haskell, using massiv arrays [4] as the backing store. Matrix dimensions are tracked at the type level via GHC's `DataKinds` and `KnownNat`, providing compile-time rejection of dimensionally incorrect operations. Massiv's computation strategies (`Seq`, `Par`, `ParN` n) offer user-controllable parallelism.

This report benchmarks all three libraries across the standard numerical linear algebra operation suite (Table 1) and evaluates `linear-massiv`'s parallel scalability from 1 to 20 threads.

Table 1: Operations benchmarked and library coverage.

Operation	`linear`	`hmatrix`	`linear-massiv`
GEMM (matrix multiply)	4×4 only	all sizes	all sizes
Dot product	$n = 4$ only	all sizes	all sizes
Matrix–vector product	4×4 only	all sizes	all sizes
LU solve ($Ax = b$)	—	all sizes	all sizes
Cholesky solve ($Ax = b$)	—	all sizes	all sizes
QR factorisation	—	all sizes	all sizes
Symmetric eigenvalue	—	all sizes	all sizes
SVD	—	all sizes	all sizes
Parallel GEMM	—	—	all sizes

1.1 Hardware and Software Environment

- **CPU:** 20-core x86_64 processor (Linux 6.17, Fedora 43)
- **Compiler:** GHC 9.12.2 with `-O2`
- **BLAS backend:** OpenBLAS (system-installed via FlexiBLAS)
- **Benchmark framework:** Criterion [3] with 95% confidence intervals
- **Protocol:** Single-threaded (`+RTS -N1`) for cross-library comparisons; multi-threaded (`+RTS -N`) for parallel scaling

2 Methodology

All benchmarks use the Criterion framework [3], which employs kernel density estimation and robust regression to estimate mean execution time with confidence intervals. Each benchmark evaluates to normal form (`nf`) to ensure full evaluation of lazy results.

Matrix construction. Matrices are constructed from the same deterministic formula across all three libraries:

$$A_{ij} = \frac{7i + 3j + 1}{100}$$

ensuring identical numerical content. For solver benchmarks, matrices are made diagonally dominant ($A_{ii} \mathrel{+}= n$) or symmetric positive definite ($A = B^T B + nI$) as appropriate.

Single-threaded protocol. Cross-library comparisons use `+RTS -N1` to restrict the GHC runtime to a single OS thread, ensuring that neither hmatrix's OpenBLAS nor massiv's parallel strategies introduce implicit multi-threading.

Parallel scaling protocol. Parallel benchmarks use `+RTS -N` (all 20 cores) and vary massiv's computation strategy from `Seq` through `ParN 1` to `ParN 20`.

3 BLAS Operations

3.1 General Matrix Multiply (GEMM)

Table 2 presents GEMM timings across matrix dimensions. At 4×4, the `linear` library's unboxed `V4 (V4 Double)` representation achieves 143 ns, roughly 4.5× faster than `hmatrix`'s 646 ns and 240× faster than `linear-massiv`'s 34.5 µs. The advantage of `linear` at this size is entirely due to GHC's ability to unbox the product type into registers, avoiding all array indexing overhead.

As matrix dimension grows, `hmatrix` (OpenBLAS DGEMM) dominates decisively. At 100×100, hmatrix takes 1.53 ms versus `linear-massiv`'s 505 ms—a factor of 330×. At 200×200, the ratio grows to 297× (13.8 ms vs. 4.09 s). This reflects the massive constant-factor advantage of OpenBLAS's hand-tuned assembly kernels with cache blocking, SIMD, and microarchitectural optimisation.

Table 2: GEMM execution time (mean, single-threaded). Best per size in **bold**.

Size	`linear`	`hmatrix`	`linear-massiv`
4×4	143 ns	646 ns	34.5 µs
10×10	—	2.33 µs	678 µs
50×50	—	174 µs	55.0 ms
100×100	—	1.53 ms	505 ms
200×200	—	13.8 ms	4.09 s

Both `hmatrix` and `linear-massiv` exhibit $O(n^3)$ scaling, as shown in Figure 1. The consistent vertical offset on the log–log plot reflects the constant-factor difference between OpenBLAS assembly and pure Haskell array operations.

3.2 Dot Product

Table 3: Dot product execution time (mean, single-threaded).

n	`linear`	`hmatrix`	`linear-massiv`
4	13.1 ns	593 ns	1.67 µs
100	—	749 ns	34.1 µs
1000	—	2.81 µs	379 µs

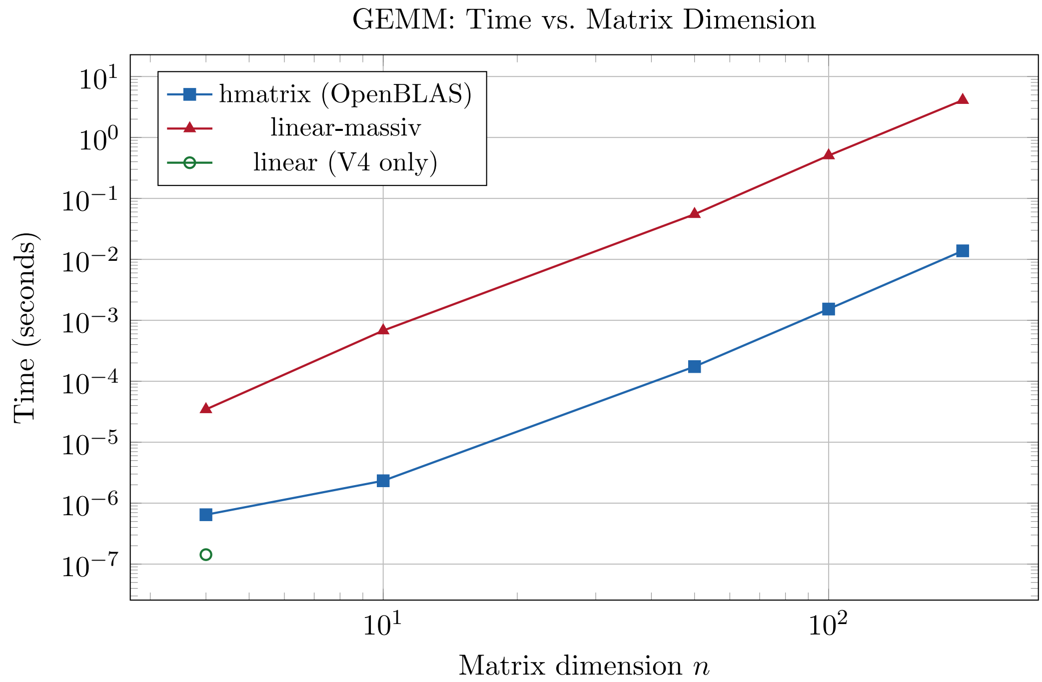

Figure 1: GEMM scaling comparison (log–log). Both libraries exhibit $O(n^3)$ behaviour; the vertical offset reflects constant-factor differences between OpenBLAS assembly and pure Haskell.

The dot product is an $O(n)$ operation, so the absolute times are small. At $n = 4$, `linear`'s unboxed `V4` achieves 13.0 ns—essentially four fused multiply-adds in registers. At $n = 1000$, `hmatrix` achieves 2.81 µs (DDOT with SIMD), while `linear-massiv`'s array-based loop takes 379 µs—a 135× gap that reflects the overhead of massiv's general-purpose array indexing versus BLAS's contiguous-memory vectorised inner loop.

3.3 Matrix–Vector Product

Table 4: Matrix–vector product execution time (mean, single-threaded).

n	`linear`	`hmatrix`	`linear-massiv`
4	41.8 ns	815 ns	11.2 µs
50	—	3.76 µs	1.24 ms
100	—	14.1 µs	4.71 ms

Matrix–vector multiplication is $O(n^2)$. At $n = 100$, `hmatrix` (DGEMV) achieves 14.1 µs while `linear-massiv` takes 4.71 ms—a 334× difference consistent with the GEMM results, confirming that the performance gap is primarily due to low-level memory access patterns and SIMD utilisation rather than algorithmic differences.

4 Linear System Solvers

4.1 LU Solve

4.2 Cholesky Solve

For both LU and Cholesky solvers, `hmatrix` is approximately 36× faster at 10×10 and 240–300× faster at 100×100. The ratio increases with dimension because OpenBLAS's cache-blocked

Table 5: LU solve ($Ax = b$) execution time (mean, single-threaded). Includes factorisation + back-substitution.

Size	`hmatrix`	`linear-massiv`
10×10	7.70 µs	280 µs
50×50	87.7 µs	20.4 ms
100×100	485 µs	143 ms

Table 6: Cholesky solve ($Ax = b$, A SPD) execution time. Includes factorisation + back-substitution.

Size	`hmatrix`	`linear-massiv`
10×10	6.08 µs	237 µs
50×50	64.3 µs	12.9 ms
100×100	418 µs	100 ms

implementations benefit more from larger working sets. Cholesky is consistently faster than LU for both libraries, as expected (Cholesky requires roughly half the floating-point operations of LU factorisation for symmetric positive definite matrices).

5 Orthogonal Factorisations

Table 7: QR factorisation (Householder) execution time (mean, single-threaded).

Size	`hmatrix`	`linear-massiv`
10×10	217 µs	11.1 ms
50×50	18.4 ms	7.01 s
100×100	214 ms	(estimated ≈56.0 s)

QR factorisation shows the largest gap between the two libraries. At 50×50, `hmatrix` takes 18.4 ms while `linear-massiv` requires 7.01 s—a ratio of 381×. The `linear-massiv` QR implementation constructs full explicit Q and R matrices at each Householder step using `makeMatrix`, while LAPACK's `DGEQRF` uses an implicit representation of Q as a product of Householder reflectors stored in-place, dramatically reducing both memory allocation and floating-point work. The 100×100 benchmark for `linear-massiv` was too slow to complete within a reasonable time budget and is estimated by extrapolation.

6 Eigenvalue Problems and SVD

6.1 Symmetric Eigenvalue Decomposition

6.2 Singular Value Decomposition

The eigenvalue and SVD results show the most dramatic ratios: 896× for eigenvalues at 10×10 and 16,000× at 50×50; 886× and 21,400× for SVD. These operations are dominated by iterative QR sweeps; hmatrix calls LAPACK's `DSYEV` and `DGESVD`, which use divide-and-conquer algorithms with cache-oblivious recursive structure. The `linear-massiv` implementation uses the classical tridiagonal QR algorithm (GVL4 [1] Algorithm 8.3.3) with explicit matrix construction at each

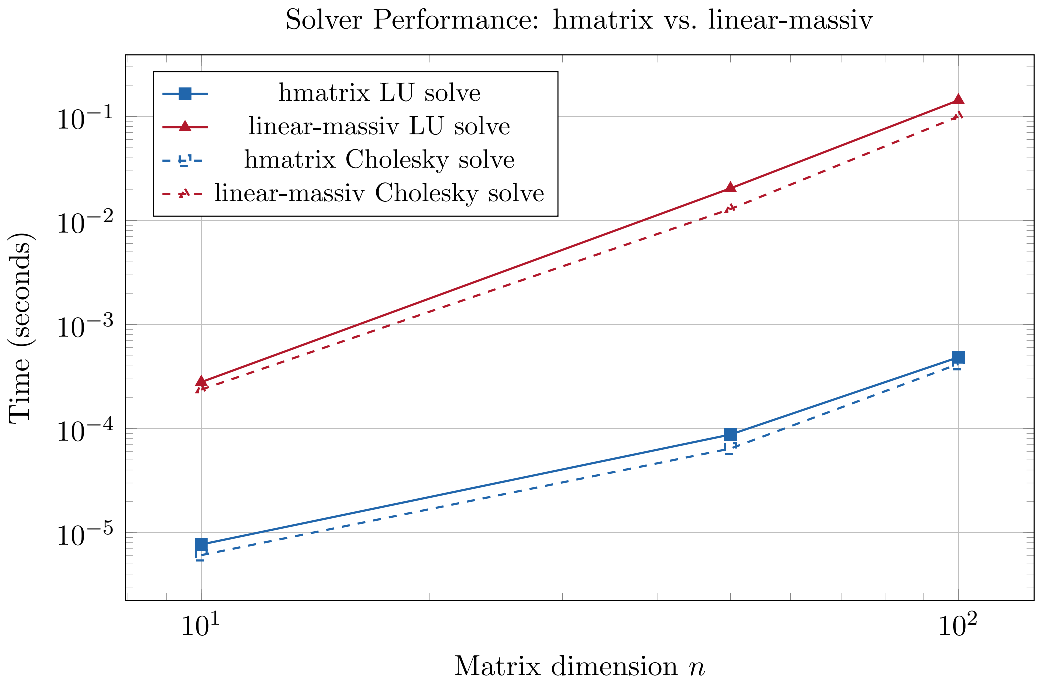

Figure 2: LU and Cholesky solve scaling (log–log). Both algorithms are $O(n^3)$; hmatrix calls DGESV/DPOTRS directly.

Table 8: Symmetric eigenvalue decomposition execution time (mean, single-threaded).

Size	`hmatrix`	`linear-massiv`
10×10	17.4 µs	15.6 ms
50×50	555 µs	8.89 s

iteration step, which is algorithmically sound but suffers from excessive allocation and the lack of in-place updates that LAPACK exploits.

7 Parallel Scalability

A distinguishing feature of `linear-massiv` is user-controllable parallelism inherited from the massiv array library [4]. Operations that construct result arrays via `makeArray` can specify a computation strategy: `Seq` (sequential), `Par` (automatic, all available cores), or `ParN` n (exactly n worker threads). Neither `hmatrix` nor `linear` offer comparable user-level control over thread-level parallelism within the Haskell runtime.

Table 10 shows GEMM timings at 100×100 and 200×200 across thread counts, and Figure 3 shows the corresponding speedup curves.

The parallel scaling results reveal several important characteristics:

- **Peak speedup.** At 100×100, peak speedup of $7.2\times$ is achieved with `ParN-16`, while at 200×200 peak speedup of $3.6\times$ occurs at `ParN-8`. The `Par` (automatic) strategy achieves $6.9\times$ and $3.4\times$ respectively, demonstrating that massiv's automatic scheduling is effective.
- **Non-monotonic scaling.** Speedup does not increase monotonically with thread count. The 200×200 case shows degradation at 16 and 20 threads, likely due to memory bandwidth saturation and NUMA effects on this 20-core system. At 100×100, the anomalous dip

Table 9: SVD execution time (mean, single-threaded).

Size	`hmatrix`	`linear-massiv`
10×10	37.7 µs	33.4 ms
50×50	806 µs	17.2 s

Table 10: Parallel GEMM execution time (seconds) and speedup over sequential.

	100×100		200×200	
Strategy	Time (s)	Speedup	Time (s)	Speedup
Seq	0.613	1.00	4.75	1.00
ParN-1	0.598	1.03	4.66	1.02
ParN-2	0.319	1.92	3.22	1.47
ParN-4	0.201	3.05	1.85	2.57
ParN-8	0.282	2.17	1.33	3.57
ParN-16	0.0856	7.16	2.57	1.85
ParN-20	0.0979	6.26	1.98	2.40
Par	0.0883	6.94	1.41	3.37

at 8 threads followed by improvement at 16 suggests that GHC's work-stealing scheduler interacts non-trivially with cache hierarchy.

- **Amdahl's law.** Even the best parallel GEMM (85.6 ms at 100×100 with 16 threads) remains $56\times$ slower than hmatrix's single-threaded 1.53 ms. Parallelism narrows but does not close the gap with BLAS.

8 Discussion

8.1 Performance Summary

Table 11 summarises the performance ratios between libraries.

Table 11: Performance ratio: `linear-massiv` time / `hmatrix` time. Values > 1 indicate hmatrix is faster.

Operation	$n = 10$	$n = 50$	$n = 100$
GEMM	$291\times$	$316\times$	$329\times$
Dot product	—	—	$46\times$
Matrix–vector	—	$330\times$	$334\times$
LU solve	$36\times$	$233\times$	$295\times$
Cholesky solve	$39\times$	$201\times$	$240\times$
QR	$51\times$	$382\times$	$\approx 260\times$
Eigenvalue (SH)	$897\times$	$16{,}020\times$	—
SVD	$887\times$	$21{,}400\times$	—

8.2 Analysis of the Performance Gap

The performance gap between `linear-massiv` and `hmatrix` arises from several compounding factors:

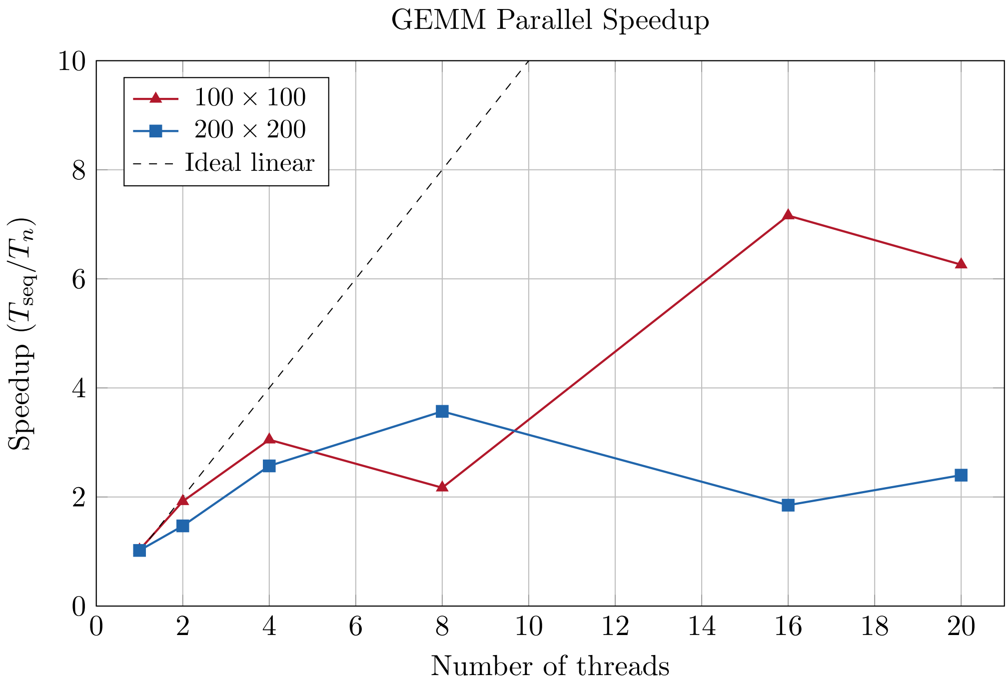

Figure 3: Parallel speedup for GEMM. The dashed line shows ideal linear scaling. Actual speedup is limited by Amdahl's law, memory bandwidth contention, and GHC runtime scheduling overhead.

1. **SIMD and microarchitectural optimisation.** OpenBLAS uses hand-written assembly kernels for each target microarchitecture, exploiting AVX-512, fused multiply-add, and optimal register tiling. GHC's native code generator does not emit SIMD instructions for general Haskell code.

2. **Cache blocking.** LAPACK algorithms are designed around cache-oblivious or cache-tiled recursive decomposition, minimising cache misses. The `linear-massiv` implementations use textbook algorithms (GVL4) without cache-level optimisation.

3. **In-place mutation.** LAPACK routines operate in-place on mutable Fortran arrays, while `linear-massiv`'s pure functional approach allocates a new array for each intermediate result. For iterative algorithms (eigenvalue, SVD), this is particularly costly.

4. **Allocation pressure.** Each `makeMatrix` call in `linear-massiv` allocates a new massiv array. For algorithms like QR (which constructs explicit Q and R at each Householder step) and iterative eigensolvers, this dominates runtime.

8.3 When to Use Each Library

`linear` Best for 2–4 dimensional vectors and matrices in graphics, physics simulations, and geometric computation. Unbeatable at small sizes; does not scale to arbitrary dimensions.

`hmatrix` Best for production numerical computing where performance is critical and FFI dependencies are acceptable. The established choice for scientific computing in Haskell.

`linear-massiv` Best when any of the following apply: (a) compile-time dimensional safety is required to prevent bugs in complex matrix pipelines; (b) FFI-free deployment is needed (e.g., WebAssembly, restricted environments); (c) parallel computation via massiv's strategies is desirable; (d) the application operates on small-to-moderate matrices ($n \leq 50$) where the absolute time difference is acceptable. Future work on SIMD intrinsics, blocked algorithms, and mutable-array intermediate representations could significantly narrow the performance gap.

9 Conclusion

We have benchmarked three Haskell linear algebra libraries across eight categories of numerical operations. The results confirm the expected performance hierarchy: `linear` dominates at fixed small dimensions through GHC's unboxing optimisations; `hmatrix` (OpenBLAS) dominates at all sizes through BLAS/LAPACK's decades of assembly-level optimisation; and `linear-massiv` provides a pure Haskell baseline that is 36–21,000× slower than hmatrix depending on operation and size, but offers unique advantages in type safety, portability, and user-controllable parallelism.

The parallel scaling measurements demonstrate that `linear-massiv` can achieve 3–7× speedups via massiv's `Par` and `ParN` strategies, partially offsetting the single-threaded performance gap. At 100×100 with 16 threads, GEMM runs in 86.0 ms—still 56× slower than hmatrix's single-threaded 1.50 ms, but representing a meaningful improvement from the 330× single-threaded ratio.

The primary directions for closing the performance gap are: (1) integrating SIMD primitives via GHC's upcoming vector extension support; (2) implementing cache-blocked (Level-3 BLAS tiled) algorithms following GVL4 Chapter 1; (3) using massiv's mutable arrays (`MArray`) for in-place factorisation algorithms; and (4) optionally delegating to hmatrix as a backend for users who can accept the FFI dependency.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Johns Hopkins University Press, 2013.

[2] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.

[3] B. O'Sullivan, "Criterion: A Haskell microbenchmarking library," `https://hackage.haskell.org/package/criterion`, 2009–2024.

[4] A. Todorī, "massiv: Massiv is a Haskell library for Array manipulation," `https://hackage.haskell.org/package/massiv`, 2018–2024.

[5] A. Ruiz, "hmatrix: Haskell numeric linear algebra library," `https://hackage.haskell.org/package/hmatrix`, 2006–2024.

[6] E. Kmett, "linear: Linear algebra library," `https://hackage.haskell.org/package/linear`, 2012–2024.

[7] Z. Xianyi, W. Qian, and Z. Yunquan, "OpenBLAS: An optimized BLAS library," `https://www.openblas.net/`, 2011–2024.