Before MPI: Locality and Kernel Policy in One-Sided Jacobi SVD

Jiarui Guo

Abstract—Parallel numerical programs often fail before communication becomes the bottleneck. This paper studies that failure boundary through one-sided Jacobi singular value decomposition (SVD), whose column-pair structure exposes where locality and kernel policy meet. We evaluate a controlled factorial space with six matrix layouts, five Jacobi kernel variants, and fourteen synthetic matrix families, measuring runtime, memory use, convergence, and reconstruction accuracy across 420 configurations. The results show that local data layout is the dominant design choice in this implementation: column-contiguous storage consistently outperforms more elaborate layouts, while dynamic ordering reduces sweeps too little to repay its observation cost. The MPI-friendly kernel exposes a useful phase dependency structure, but the measured MPI runs should be read as wrapper overhead rather than evidence of distributed SVD scalability. The main lesson is simple: before scaling Jacobi SVD across ranks, the local memory trajectory and kernel policy cost model must already be right.

Index Terms—Singular value decomposition, one-sided Jacobi method, matrix layout, kernel policy, performance evaluation, Message Passing Interface.

I. Introduction

Singular value decomposition (SVD) is a central primitive in numerical linear algebra, machine learning, and scientific computing. In principle, SVD is a solved primitive; in practice, an implementation is never just its arithmetic count. The same mathematical routine can become a streaming kernel, a cache-hostile traversal, a synchronization bottleneck, or an MPI wrapper whose measured cost has little to do with useful distributed work. For parallel numerical code, this distinction is not cosmetic. It is the difference between scaling an algorithm and scaling an accident.

This paper studies that boundary through one-sided Jacobi SVD. Jacobi is not chosen here to compete with production bidiagonalization-based SVD solvers [1]–[3]. It is chosen because its structure is unusually exposed: the computation is a sequence of column-pair orthogonalizations. Those pairs are exactly where memory layout, pivot scheduling, thread-level execution, and future rank-level communication meet. Jacobi therefore gives a compact laboratory for a broader systems question: before distributing an algorithm with MPI, which local design choices already determine its behavior?

The central claim of this work is deliberately modest but sharp. For the tested SVD implementation, local data layout dominates the more elaborate scheduling choices. Column-contiguous storage gives the Jacobi kernel the access pattern it wants; more sophisticated layouts and dynamic schedules can add overhead without buying enough convergence or reuse. The MPI-friendly path is valuable as a dependency structure, but it should not be mistaken for evidence of distributed SVD scalability. This distinction echoes the larger lesson of communication-conscious numerical linear algebra: data movement is a first-class cost, not an afterthought [4], [5].

We make this point through a controlled factorial study. The mathematical operation is fixed, while the implementation varies matrix layout, Jacobi kernel strategy, and input workload. This lets the experiment separate effects that are often collapsed into one wall-clock number: locality, convergence, execution overhead, memory footprint, and reconstruction accuracy.

The paper makes the following contributions:

- a layout-parametric one-sided Jacobi SVD implementation with row-major, column-major, strided, jagged, blocked, and Morton-order matrix representations;
- a family of Jacobi kernels covering cyclic, correlation-ordered, phase-scheduled, Pthreads-parallel, and SIMD-oriented execution strategies;
- a reproducible full-factorial benchmark over matrix families, layouts, and kernels, with measurements for timing, memory consumption, convergence, and reconstruction quality;
- an analysis framework that separates locality effects, scheduling effects, and execution overheads before claiming rank-level distributed SVD performance;
- empirical evidence that, within this design space, column-contiguous storage is the dominant local choice, while global dynamic ordering and MPI wrapper measurements require narrower interpretation.

The remainder of this paper is organized as follows. Section II introduces the technical background. Section III presents the method. Section IV reports the experiments. Section V discusses limitations and implications. Section VI concludes the paper.

II. Background and Related Work

This section positions the study within four related lines of work: SVD algorithms and Jacobi pivoting, architecture-sensitive matrix layout, matrix properties used for numerical stress, and parallel SVD across vector, thread, and message-passing machines.

Jiarui Guo is with Nankai University. E-mail: guojiarui@mail.nankai.edu.cn.

A. SVD Algorithms and One-Sided Jacobi Structure

For a tall matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, the thin singular value decomposition (SVD) writes

$$A = U\Sigma V^T, \tag{1}$$

where $U \in \mathbb{R}^{m\times n}$ has orthonormal columns, Σ is diagonal with nonnegative singular values, and $V \in \mathbb{R}^{n\times n}$ is orthogonal. Classical dense SVD software is dominated by bidiagonalization-based algorithms: the Golub–Reinsch line first reduces A to bidiagonal form and then solves the bidiagonal SVD problem [1], [2]. LAPACK exposes this family through drivers such as xGESVD and xGESDD, the latter using a divide-and-conquer stage for the bidiagonal problem when singular vectors are requested [3]. Krylov and Lanczos bidiagonalization methods are natural when only a few singular triplets of a large sparse or matrix-free problem are needed, while randomized SVD is often attractive for low-rank approximation because it replaces a full factorization with a small number of matrix passes and a compressed subspace problem [6].

Those alternatives are not the target of this study. The implementation computes a full thin SVD of moderate dense matrices and intentionally exposes the column-pair structure that production bidiagonalization routines largely hide. One-sided Jacobi SVD, rooted in Hestenes' biorthogonalization viewpoint [7], iterates over column pairs of a working matrix G initialized from A. For a pair (i, j), the kernel forms

$$\alpha = g_i^T g_i, \qquad \beta = g_j^T g_j, \qquad \gamma = g_i^T g_j, \tag{2}$$

and applies a plane rotation when

$$\frac{|\gamma|}{\sqrt{\alpha\beta}} > \varepsilon. \tag{3}$$

The same rotation is accumulated into V. At convergence, the column norms of G are the singular values and the normalized columns of G form U. This formulation gives a compact numerical kernel: two column dot products, one cross product, and two two-column updates.

The main reason to choose one-sided Jacobi in this project is parallelizability, not asymptotic operation count. Bidiagonalization methods are typically the right default for highly tuned dense libraries, and Jacobi methods also have an important accuracy tradition: Demmel and Veselić showed that Jacobi-type methods can compute small singular information with high relative accuracy under suitable conditions, and Drmač and Veselić later developed fast and accurate preconditioned Jacobi SVD algorithms [8], [9]. In this paper, however, that accuracy record is secondary. The decisive property is structural: one-sided Jacobi turns SVD into a sequence of local column orthogonalizations. A Jacobi sweep consists of all $\binom{n}{2}$ column pairs; the chosen pivot ordering controls both convergence behavior and the amount of parallel work exposed at each step.

Pivot ordering is therefore part of the algorithm, not a cosmetic loop permutation. A cyclic ordering has low scheduling overhead but does not prioritize the largest remaining correlations. Dynamic orderings try to remove the most nonorthogonal pairs earlier, trading possible sweep reductions for the cost of rescoring and reordering pairs. Parallel orderings replace a total order by phases: each phase is a matching of the complete graph on the n columns, so no two pairs in the same phase share a column. Such orderings have long been used for vector and parallel Jacobi implementations [10], [11]. This makes one-sided Jacobi a good laboratory for separating convergence, locality, and scheduling effects before moving to a distributed column-block design.

B. Memory Layout as an Algorithmic Parameter

The arithmetic count of one Jacobi sweep hides the memory system. Each rotation streams two columns of G and two columns of V. On a modern CPU, this means the actual cost is shaped by cache-line utilization, hardware prefetching, translation lookaside buffer behavior, memory bandwidth, and coherence traffic, not only by the number of multiply-adds. This architecture dependence has been recognized for decades in numerical linear algebra: Gallivan et al. argued that hierarchical memory systems force linear algebra algorithms to be designed around locality, and Lam, Rothberg, and Wolf showed that blocked algorithms can be understood through their cache behavior and block sizes [12], [13]. Drepper gives the systems-level version of the same lesson: sequential, predictable memory streams exploit spatial locality, while large strides and irregular access patterns waste cache lines and defeat prefetching [14].

Dense linear algebra libraries respond to this architecture by changing the algorithmic form. LAPACK turns many factorizations into blocked algorithms so that most work is performed by Level-3 BLAS; Goto and van de Geijn's analysis of high-performance matrix multiplication makes the same point more sharply by organizing packing, cache blocking, and a small inner kernel around the memory hierarchy [3], [15]. Cache-oblivious work takes a different route: recursive algorithms and recursive layouts try to obtain locality across cache levels without hard-coding a particular cache size [16]. These results make layout an algorithmic concern, not just a container choice.

One-sided Jacobi SVD is especially sensitive to this issue because the hot loop is a column stream. If a logical column is contiguous, dot products and rotations can be served as mostly sequential loads and stores. If a logical column is strided, each useful value may arrive with nearby values that the column kernel does not immediately use. Padding, indirection, blocking, and space-filling coordinate maps all change the balance among useful bandwidth, cache reuse, and indexing overhead. Thus the same Jacobi sweep can exercise very different parts of the memory hierarchy depending on the mapping from matrix coordinates to addresses.

C. Matrix Properties and Numerical Stress

Synthetic matrices are useful only when they control properties that matter to an SVD algorithm. The relevant

axes are not the names of distributions but the singular spectrum, conditioning, numerical rank, scaling, sparsity, and column correlation. These properties determine what the algorithm must resolve, how quickly iterative variants converge, and which finite-precision effects are likely to become visible.

The singular spectrum is the central axis. Fast spectral decay or a clear numerical rank makes low-rank approximation effective, while a flat spectrum or a cluster of nearly equal singular values makes subspace separation fragile. Perturbation theory gives the reason: singular values are stable in an absolute sense, but singular vectors and singular subspaces depend on separation gaps. Wedin-style bounds and later SVD perturbation surveys make this gap dependence explicit, showing why small residuals do not necessarily imply well-determined singular vectors when neighboring singular values are close [17]–[19]. For Krylov, Lanczos, and randomized SVD methods, the same spectrum controls convergence because spectral gaps and decay rates determine how rapidly the desired subspace is isolated [6].

Conditioning and scaling form a second axis. The 2-norm condition number is the ratio $\sigma_{\max}/\sigma_{\min}$, so small singular values are the part of the factorization most exposed to relative error. Ill-conditioned test matrices therefore probe a different failure mode from ordinary dense random matrices: they stress whether the method can preserve information in small singular directions rather than merely reduce a Frobenius-norm residual. Random-matrix theory gives a useful baseline for unstructured Gaussian inputs, including the distribution of condition numbers and extreme singular values [20]; classical numerical analysis texts use highly ill-conditioned examples, such as Hilbert-type matrices, to expose the limits of finite-precision algorithms [21].

Rank structure, sparsity, and scale heterogeneity change the work seen by Jacobi-style algorithms. Low-rank and discrete ill-posed problems often have gradually decaying singular values, making the notion of an exact cutoff rank ambiguous [22]. Sparse or geometrically local matrices alter column correlations and can create many pairs that are already close to orthogonal. Heavy-tailed or strongly scaled data can make a few column norms dominate the normalized correlation tests used by one-sided Jacobi. Thus a benchmark suite should be interpreted as a set of controlled perturbations of algorithmically relevant properties: column norms, pairwise correlations, zero or near-zero directions, spectral gaps, decay rates, and condition number.

D. Vector, Thread, and Message-Passing Parallel SVD

Parallel SVD research spans several granularities, and Jacobi methods are attractive because their column-pair structure maps naturally onto more than one of them. At the finest granularity, SIMD (single instruction, multiple data) vectorization targets the dot products, norms, and two-column updates inside each rotation. This is not merely a compiler detail: thread-parallel Jacobi SVD work has shown that vector-friendly data layout and explicit SIMD instructions can be central to efficient Jacobi kernels [23]. In other words, vectorization and matrix representation meet exactly at the column stream.

At the shared-memory granularity, the relevant unit is no longer one arithmetic operation but a set of independent column pairs. If a pivot strategy forms phases whose pairs do not share columns, those pairs can be assigned to worker threads without write-write conflicts on the working matrix or on the accumulated right singular vectors. This is the intersection between Pthreads-style programming and Jacobi SVD: Pthreads provides a portable thread substrate, but the SVD algorithm supplies the dependency structure that makes thread-level work safe. The hard part is therefore not calling a threading API; it is choosing a pivot schedule whose parallel work is large enough to amortize thread-pool and synchronization overhead.

At the distributed-memory granularity, MPI provides the standard message-passing model for process-level parallel programs [24]. Distributed SVD algorithms must decide how to partition matrix columns or blocks, when to exchange data, and how to synchronize global orthogonalization progress. Studies of large-scale SVD optimization emphasize that the bottleneck often shifts among local kernels, communication, synchronization, and memory movement as the architecture changes [25]. For one-sided Jacobi specifically, a round-robin or block-Jacobi phase schedule is the natural bridge from local rotations to rank-level work: independent pairs can be processed concurrently, while phase boundaries identify where communication or synchronization would be needed. The broader lesson for the present study is that parallel SVD performance should be attributed across computation, data movement, scheduling, and synchronization rather than to a single notion of "parallelism."

III. Method and Design

This section describes the method as a set of deliberately separated design axes. The mathematical algorithm fixes the invariant operation: repeated orthogonalization of column pairs in a one-sided Jacobi SVD. The kernel layer changes the ordering and execution substrate of that operation. The matrix layer changes only the coordinate-to-address mapping. The profiling layer then measures the resulting executions through the same benchmark workflow. This separation is important because a fast SVD implementation can fail for several different reasons: too many sweeps, poor spatial locality, excessive allocation traffic, synchronization overhead, or simply an unsuitable measurement method.

A. One-Sided Jacobi Principle and Cost Model

Let $A \in \mathbb{R}^{m\times n}$, with $m \geq n$. The implementation computes a thin SVD,

$$A = U\Sigma V^T, \tag{4}$$

by initializing a working matrix $G \leftarrow A$ and a right singular vector accumulator $V \leftarrow I_n$. A Jacobi step chooses two columns g_i and g_j of G and computes

$$\alpha = g_i^T g_i, \qquad \beta = g_j^T g_j, \qquad \gamma = g_i^T g_j. \tag{5}$$

If either column is zero, or if the normalized correlation

$$\rho_{ij} = \frac{|\gamma|}{\sqrt{\alpha\beta}} \tag{6}$$

is no larger than the tolerance ε, the pair is already sufficiently orthogonal and no rotation is applied. Otherwise the algorithm chooses a plane rotation

$$J = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{7}$$

such that the two updated columns become orthogonal. In the stable tangent formulation used by the implementation,

$$\tau = \frac{\beta - \alpha}{2\gamma}, \qquad t = \frac{\operatorname{sign}(\tau)}{|\tau| + \sqrt{1+\tau^2}}, \tag{8}$$

and therefore

$$c = \frac{1}{\sqrt{1+t^2}}, \qquad s = ct. \tag{9}$$

The same rotation is applied to the pair of columns in G and accumulated into V. When a full sweep produces no rotations, the algorithm reports convergence. The singular values are then the column norms $\sigma_i = \|g_i\|_2$, and the nonzero left singular vectors are obtained by $u_i = g_i/\sigma_i$. Finally, singular triplets are sorted in non-increasing singular-value order.

For one column pair, the dominant arithmetic cost is linear in m: two column norms, one cross product, and a two-column update of G. Accumulating the same rotation into V costs $O(n)$. Thus one active pair costs $O(m+n)$ arithmetic operations, and a complete cyclic sweep over $\binom{n}{2}$ pairs costs

$$O\big(n^2(m+n)\big). \tag{10}$$

For the tall matrices used in the evaluation, this is effectively $O(mn^2)$ per sweep. If the method takes S sweeps, the overall iterative cost is

$$O\big(Sn^2(m+n)\big), \tag{11}$$

plus $O(mn)$ for copying the input and forming U, $O(n^2)$ for initializing and storing V, and $O(n \log n)$ index sorting followed by $O(mn+n^2)$ data movement for reordering the triplets.

This cost model is intentionally incomplete in one respect: it counts arithmetic but not locality. Jacobi SVD streams columns. If the layout makes a logical column contiguous, the inner products and rotations are mostly sequential memory accesses. If a column is represented by large strides, indirection, or a recursive coordinate map, the same asymptotic sweep may produce very different cache and allocation behavior. The rest of this section therefore treats scheduling and layout as first-class methodological variables rather than as coding details.

B. KernelLike SVD Variants

All five kernels share the same rotation operator from Section III. The implementation treats the algorithmic design space as a small algebraic data type: scheduling, execution, and local numeric behavior are independent finite choices, and a concrete SVD kernel is their product. In notation,

$$\mathfrak{K} = \Pi \times X \times \Omega, \tag{12}$$

where Π is the set of pivot schedules, X is the set of execution policies, and Ω is the set of inner-kernel policies. Template parameterization realizes one element of $\mathfrak{K}$ at compile time. This preserves the mathematical shape of the design space while avoiding run-time dispatch on the inner Jacobi path.

We therefore separate a concrete kernel into two orthogonal design layers. The first layer decides which column pairs are exposed to the Jacobi operator; the second layer decides how the exposed independent work is executed. Formally, a kernel is a stateless triple

$$\mathcal{K} = (\pi, \chi, \omega), \tag{13}$$

where π is the pivot schedule, χ is the execution policy, and ω is the numerical inner-kernel policy. The input state is only

$$(G^{(t)}, V^{(t)}, \varepsilon, S_{\max}), \tag{14}$$

so changing $\mathcal{K}$ does not introduce hidden mutable state. This is the relevant KernelLike-style constraint for the whole-matrix SVD kernels: the C++ concept named KernelLike is stricter and describes coordinate pure functions, but the SVD variants follow the same stateless-object design discipline.

1) Pivot scheduling: Let the complete set of Jacobi column pairs be

$$\mathcal{P}_n = \{(i,j) : 0 \le i < j < n\}, \qquad |\mathcal{P}_n| = \binom{n}{2}. \tag{15}$$

The cyclic schedule is the fixed lexicographic sequence

$$\pi_{\text{cyc}} = \big((0,1),(0,2),\ldots,(0,n-1),(1,2),\ldots,(n-2,n-1)\big). \tag{16}$$

It has no per-sweep scoring cost and therefore represents the reference schedule.

The dynamic schedule recomputes the pivot order at the beginning of sweep t. For each pair,

$$s_t(i,j) = \frac{|(g_i^{(t)})^T g_j^{(t)}|}{\|g_i^{(t)}\|_2 \|g_j^{(t)}\|_2}, \tag{17}$$

with score 0 when either norm is zero. The schedule is then

$$\pi_{\text{dyn}}^{(t)} = \operatorname{argsort}^{\downarrow}_{(i,j)\in\mathcal{P}_n} s_t(i,j). \tag{18}$$

The intended benefit is faster removal of the largest off-diagonal Gram entries; the explicit cost is rescoring all pairs and sorting, i.e. an additional scheduling term of $O(n^2 m + n^2 \log n)$ per sweep.

The phase schedule partitions $\mathcal{P}_n$ into disjoint matchings

$$\Pi_{\mathrm{rr}} = (\Phi_1, \ldots, \Phi_q), \qquad \bigcup_{k=1}^{q} \Phi_k = \mathcal{P}_n, \tag{19}$$

such that

$$(i,j),(p,r) \in \Phi_k,\ (i,j) \neq (p,r) \Longrightarrow \{i,j\} \cap \{p,r\} = \varnothing. \tag{20}$$

Thus every pair inside one phase can rotate without a write-write conflict on G or V. The round-robin construction gives $q = n-1$ phases for even n and $q = n$ phases for odd n, with at most $\lfloor n/2 \rfloor$ pairs per phase. This is the scheduling abstraction used both by the MPI-friendly variant and by the Pthreads variant.

2) Execution and inner-kernel policy: The execution policy is

$$\chi \in \{\chi_{\mathrm{serial}}, \chi_{\mathrm{pthreads}}\}, \tag{21}$$

where χ_{serial} applies scheduled pairs sequentially, while χ_{pthreads} applies all pairs in one phase concurrently:

$$\chi_{\mathrm{pthreads}}(\Phi_k) = \mathrm{parallel\_for}_{(i,j)\in\Phi_k}\, \mathcal{J}_{ij}. \tag{22}$$

The phase-disjointness condition above is what makes this expression race-free: no two rotations in Φ_k update the same column of G or V.

The numeric policy is

$$\omega \in \{\omega_{\mathrm{scalar}}, \omega_{\mathrm{simd}}\}. \tag{23}$$

It controls the local reductions and column updates inside one Jacobi operator $\mathcal{J}_{ij}$. In the current implementation, ω_{simd} is a dispatch boundary with scalar fallback, so its results should be read as the integration behavior of a SIMD-oriented path rather than as a fully vectorized microkernel. Therefore scheduling changes the order and independence structure of rotations, whereas execution and numeric policies change the mapping from that structure to hardware resources.

The five evaluated kernels are therefore

$$\mathcal{K}_{\mathrm{naive}} = (\pi_{\mathrm{cyc}}, \chi_{\mathrm{serial}}, \omega_{\mathrm{scalar}}), \tag{24}$$
$$\mathcal{K}_{\mathrm{advanced}} = (\pi_{\mathrm{dyn}}, \chi_{\mathrm{serial}}, \omega_{\mathrm{scalar}}), \tag{25}$$
$$\mathcal{K}_{\mathrm{mpi}} = (\Pi_{\mathrm{rr}}, \chi_{\mathrm{serial}}, \omega_{\mathrm{scalar}}), \tag{26}$$
$$\mathcal{K}_{\mathrm{pthreads}} = (\Pi_{\mathrm{rr}}, \chi_{\mathrm{pthreads}}, \omega_{\mathrm{scalar}}), \tag{27}$$
$$\mathcal{K}_{\mathrm{simd}} = (\pi_{\mathrm{cyc}}, \chi_{\mathrm{serial}}, \omega_{\mathrm{simd}}). \tag{28}$$

The MPI-friendly variant is intentionally only phase-scheduled in this implementation; it exposes rank-mappable independence but does not distribute Jacobi rotations across MPI ranks.

C. Allocator-Backed MatrixLike Layout Design

A MatrixLike object separates the semantic matrix from its physical realization. The logical object is the coordinate map

$$M : \{0, \ldots, n-1\} \times \{0, \ldots, m-1\} \to \mathbb{R}. \tag{29}$$

The layout is the additional structure that realizes this map. For a single-buffer layout, the realization factors through an address map

$$M(x,y) = B[\lambda(x,y)], \tag{30}$$
$$\lambda : \mathcal{D}_{n,m} \to \mathcal{A}_C, \tag{31}$$
$$\mathcal{D}_{n,m} = \{0, \ldots, n-1\} \times \{0, \ldots, m-1\}, \tag{32}$$
$$\mathcal{A}_C = \{0, \ldots, C-1\}. \tag{33}$$

where B is allocator-owned storage and C is physical capacity. For indirect layouts, the realization is a finite collection of allocator-owned buffers and a coordinate evaluator returning a buffer-offset pair. Thus all layouts implement the same $M(x,y)$ operation; they differ only in the mapping structure used to reach the scalar. This is the key experimental control: the Jacobi algorithm is unchanged, while locality, padding, allocation cardinality, and index computation are changed through the map.

1) Affine direct maps: The simplest single-buffer family uses an affine address map

$$\lambda_{\alpha,\beta}(x,y) = \alpha x + \beta y, \qquad C = mn. \tag{34}$$

The two compact dense layouts are the two canonical choices

$$\lambda_{\mathrm{row}}(x,y) = yn + x, \tag{35}$$
$$\lambda_{\mathrm{col}}(x,y) = xm + y. \tag{36}$$

Their difference is completely captured by the stride vector of λ. For a fixed column x, Jacobi reductions stream through

$$(x,0),(x,1),\ldots,(x,m-1), \tag{37}$$

so the relevant physical increment is

$$\Delta_{\mathrm{row}} = n, \qquad \Delta_{\mathrm{col}} = 1. \tag{38}$$

Row-major and column-major therefore form a controlled pair: they have the same capacity and one allocation, but they exchange the unit-stride direction.

2) Pitched affine maps: The strided layout remains affine, but replaces the logical row width by a physical pitch p:

$$p = \max(n, p_0), \qquad \lambda_{\mathrm{stride}}(x,y) = yp + x, \qquad C = mp. \tag{39}$$

Compared with compact row-major storage, the map changes only the y coefficient from n to p. The resulting padding is

$$C - mn = m(p-n), \tag{40}$$

and the column-stream increment is $\Delta_{\mathrm{stride}} = p$. This isolates leading-dimension effects: the evaluator is still a single affine expression and the allocation count is still one, but the physical distance between consecutive logical rows is no longer determined by n.

3) Indirect row maps: The jagged layout replaces the scalar address map by a two-stage evaluator. Let

$$\mathcal{B} = (B_0, B_1, \ldots, B_{m-1}) \tag{41}$$

be independently allocated row buffers. Then

$$M(x, y) = B_y[x], \qquad y = 0, \ldots, m-1, \tag{42}$$

with a separately allocated handle array storing $\mathcal{B}$. The allocation count is $m + 1$. For a fixed column x, the coordinate sequence is no longer a linear walk through one buffer:

$$B_0[x], B_1[x], \ldots, B_{m-1}[x], \tag{43}$$

so each logical step may include a handle load and a jump to a different allocation. This layout is therefore the non-affine control for pointer chasing and allocator fragmentation.

4) Locality-transforming maps: The remaining layouts keep one scalar buffer but choose non-affine maps that transform two-dimensional locality. The blocked map first decomposes a coordinate into tile and intra-tile coordinates:

$$X = \left\lfloor \frac{x}{b_x} \right\rfloor, \quad Y = \left\lfloor \frac{y}{b_y} \right\rfloor, \quad T_x = \left\lceil \frac{n}{b_x} \right\rceil. \tag{44}$$

Its address map is

$$\lambda_{\mathrm{blk}}(x, y) = (YT_x + X)b_x b_y + (y \bmod b_y)b_x + (x \bmod b_x), \tag{45}$$

with physical capacity

$$C_{\mathrm{blk}} = \left\lceil \frac{n}{b_x} \right\rceil \left\lceil \frac{m}{b_y} \right\rceil b_x b_y. \tag{46}$$

The map is piecewise affine: linear inside each tile and discontinuous at tile boundaries. It pays for divisions, moduli, and boundary padding in exchange for bounded two-dimensional reuse.

The Morton map embeds the logical rectangle in a power-of-two square

$$s = 2^{\lceil \log_2(\max(m,n)) \rceil}, \qquad C_{\mathrm{morton}} = s^2, \tag{47}$$

and uses bit interleaving:

$$\lambda_{\mathrm{morton}}(x, y) = \mathrm{bitzip}(x, y), \tag{48}$$

where low-order bits of x and y are alternated. Its padding overhead is $s^2 - mn$. Unlike the blocked map, Morton order does not choose a single tile size; it recursively interleaves neighborhoods at multiple scales. Its cost is a more expensive coordinate transform, and its benefit, if any, must come from preserving two-dimensional locality across levels of the cache hierarchy.

The exposed layouts are therefore not separate algorithms. They are six realizations of four mapping principles: compact affine maps, pitched affine maps, indirect row maps, and locality-transforming maps. Because the SVD kernels access matrices only through the MatrixLike coordinate operation, any measured layout effect must be attributable to these mapping choices.

D. Numerical Correctness and SVD Convergence

For each input matrix $A \in \mathbb{R}^{m \times n}$, a kernel returns $(\widehat{U}, \widehat{\sigma}, \widehat{V})$ with

$$\widehat{\Sigma} = \mathrm{diag}(\widehat{\sigma}_1, \ldots, \widehat{\sigma}_n), \qquad \widehat{A} = \widehat{U}\widehat{\Sigma}\widehat{V}^T. \tag{49}$$

All correctness metrics are computed from the elementwise residual

$$E = \widehat{A} - A, \qquad e_{xy} = \widehat{a}_{xy} - a_{xy}. \tag{50}$$

Let

$$\mathcal{F} = \{(x, y) : a_{xy} \in \mathbb{R},\ \widehat{a}_{xy} \in \mathbb{R}\}, \qquad N_f = |\mathcal{F}|, \tag{51}$$

and let all sums below range over $\mathcal{F}$. Non-finite pairs are counted separately as N_{nan}, N_{inf}, and $N_{\mathrm{nonfinite}}$.

The reconstruction-error family is

$$L_1(E) = \sum |e_{xy}|, \tag{52}$$

$$L_2(E) = \left(\sum |e_{xy}|^2 \right)^{1/2}, \tag{53}$$

$$L_\infty(E) = \max |e_{xy}|, \tag{54}$$

$$\mathrm{MAE}(E) = \frac{1}{N_f} \sum |e_{xy}|, \tag{55}$$

$$\mathrm{MSE}(E) = \frac{1}{N_f} \sum |e_{xy}|^2, \tag{56}$$

$$\mathrm{RMSE}(E) = \sqrt{\mathrm{MSE}(E)}. \tag{57}$$

Relative norm errors use the corresponding flattened reference norms:

$$r_p(E, A) = \frac{L_p(E)}{L_p(A)}, \qquad p \in \{1, 2, \infty\}. \tag{58}$$

The elementwise relative-error statistics use a denominator floor δ:

$$q_{xy} = \frac{|e_{xy}|}{\max(|a_{xy}|, \delta)}, \tag{59}$$

$$q_{\max} = \max q_{xy}, \qquad q_{\mathrm{mean}} = \frac{1}{N_f} \sum q_{xy}. \tag{60}$$

For dynamic range $R = \max_{\mathcal{F}} a_{xy} - \min_{\mathcal{F}} a_{xy}$, the scale-sensitive image-style summaries are

$$\mathrm{NRMSE} = \frac{\mathrm{RMSE}}{|R|}, \qquad \mathrm{PSNR} = 20 \log_{10} \frac{|R|}{\mathrm{RMSE}}. \tag{61}$$

Bit-level proximity is tracked by exact equality count and

$$d_{\mathrm{ULP}}^{\max} = \max_{(x,y) \in \mathcal{F}} d_{\mathrm{ULP}}(a_{xy}, \widehat{a}_{xy}), \tag{62}$$

where d_{ULP} is computed after mapping IEEE double values to monotone ordered bit patterns. The metric is diagnostic rather than normative: near zero, a small absolute error can still imply a large ULP distance.

Convergence is evaluated independently from reconstruction. Define the nonzero singular-index set

$$\mathcal{I}_+ = \{i : \widehat{\sigma}_i > \sigma_{\mathrm{floor}}\}, \tag{63}$$

and the compared left-vector pair set

$$\mathcal{P}_U = \{(i, j) : i < j,\ i, j \in \mathcal{I}_+\}. \tag{64}$$

The a posteriori left-orthogonality residuals are

$$\eta_{ij}^{U} = |\widehat{u}_i^T \widehat{u}_j|, \qquad (i,j) \in \mathcal{P}_U, \tag{65}$$

$$\eta_{\max}^{U} = \max_{(i,j)\in\mathcal{P}_U} \eta_{ij}^{U}, \tag{66}$$

$$\eta_{\mathrm{rms}}^{U} = \left(\frac{1}{|\mathcal{P}_U|} \sum_{(i,j)\in\mathcal{P}_U} (\eta_{ij}^{U})^2 \right)^{1/2}, \tag{67}$$

$$\nu_U = |\{(i,j) \in \mathcal{P}_U : \eta_{ij}^{U} > \varepsilon\}|. \tag{68}$$

The corresponding Gram-matrix off-diagonal residual is

$$\| \operatorname{offdiag}(\widehat{U}^T \widehat{U}) \|_F = \left(2 \sum_{(i,j)\in\mathcal{P}_U} (\eta_{ij}^{U})^2 \right)^{1/2}. \tag{69}$$

For the right singular vectors, all column pairs are checked:

$$\| \operatorname{offdiag}(\widehat{V}^T \widehat{V}) \|_F = \left(2 \sum_{1\le i<j\le n} |\widehat{v}_i^T \widehat{v}_j|^2 \right)^{1/2}. \tag{70}$$

The final convergence predicate is the conjunction

$$\text{converged} = \text{reported\_converged} \wedge (\nu_U = 0). \tag{71}$$

The report also records the sweep count S, budget $S_{\max}$, and utilization

$$\phi_{\text{sweep}} = \frac{S}{S_{\max}}. \tag{72}$$

Thus the benchmark distinguishes three notions that are often conflated: small reconstruction residual, posterior orthogonality, and termination within the Jacobi sweep budget.

E. Profiling Workflow

The benchmark workflow is instance driven. A JSON instance specifies a finite collection of samples and jobs,

$$\mathcal{I} = (\mathcal{S}, \mathcal{J}), \qquad \mathcal{J} = \{J_1, \ldots, J_q\}. \tag{73}$$

Each job binds one sample to one SVD configuration and to a sequence of profiling experiments:

$$J_k = (s_k, \ell_k, \mathcal{K}_k, \varepsilon_k, S_{\max,k}, E_k), \qquad E_k = (e_{k1}, \ldots, e_{kp_k}). \tag{74}$$

Here s_k selects the matrix sample, ℓ_k the layout, $\mathcal{K}_k$ the kernel, and E_k the ordered profiler list. Samples are generated or reused before measurement; jobs may run concurrently, but experiments inside one job run sequentially so that each profiler receives an isolated output directory. The command template expands paths such as the sample file, result directory, metrics file, layout, kernel, sweep budget, and tolerance. Each run writes ordinary stdout and stderr, profiler outputs, and the program's own JSONL metrics containing input shape, layout, kernel, reconstruction error, sweep count, convergence flags, and final orthogonality measures.

Timing is measured at two levels. GNU time records process-level resource data such as elapsed time, user time, system time, and maximum resident set size. Hyperfine provides statistically clean wall-clock timing with warmup and repeated measured runs; in our factorial timing sweep, each configuration uses warmup runs followed by repeated measurements and exports machine-readable JSON. The two tools answer different questions. GNU time is useful for coarse resource accounting, while Hyperfine is better for comparing steady-state runtimes under repeated execution.

Instruction and cache behavior are measured through Valgrind's Callgrind and Cachegrind tools. Callgrind records call-path-sensitive instruction costs and, with the selected options, jump and instruction-level detail. Cachegrind simulates cache and branch behavior, allowing us to compare whether a layout or schedule changes the memory access profile even when wall-clock timing is noisy. These tools slow the program substantially, so the profiler instances are focused rather than full factorial: they select representative matrix families, layouts, and kernels that isolate baseline dense behavior, dynamic-ordering overhead, Morton-index overhead, and degenerate zero-input behavior.

Heap behavior is measured with Heaptrack, Massif, and DHAT. Heaptrack attributes allocation hot spots and retained memory to call paths. Massif reports the evolution of heap size over time and is configured with byte-based time so that allocation volume, rather than instruction count, drives the snapshots. DHAT complements both by describing allocation lifetime and access behavior. Together, these tools are well matched to the allocator-backed layout design: compact layouts should have a small number of large allocations, jagged layouts should expose many row allocations, and phase-parallel kernels may reveal thread-pool or temporary-buffer overhead.

MPI-specific profiling is isolated to mpiP. The benchmark runner can launch a job through mpirun, enable the program's MPI execution scope, and preload mpiP when the library is available. The mpiP instances run the MPI-friendly kernel with multiple process counts and selected layouts. Since the current SVD rotations are not distributed across ranks, these runs are interpreted narrowly: they measure MPI scope, launch, barrier, and output-ownership effects around the phase-scheduled local kernel, not the scalability of a distributed SVD solver.

Finally, although the runner contains support for Linux perf wrappers, this study does not report perf counter data. The experiments were conducted under WSL2, where access to hardware performance monitoring units and kernel-level counter semantics can be unavailable, restricted, or not comparable to a native Linux installation. Rather than mix unreliable hardware counters with deterministic tool outputs, we use GNU time, Hyperfine, Valgrind-family profilers, Heaptrack, and mpiP as the measurement basis.

IV. Evaluation

The evaluation is organized around the three separable effects introduced in Section III: physical layout, pivot and execution policy, and workload structure. This organization is deliberate. A raw ranking of all 420 configurations

would mix address arithmetic, cache locality, Jacobi sweep count, thread overhead, and matrix conditioning into one number. Instead, we first establish that the benchmark completed correctly, then isolate each factor's dominant contribution.

A. Experimental Setup

The main factorial experiment used 14 matrix families, 6 MatrixLike layouts, and 5 one-sided Jacobi SVD kernels, giving $14 \times 6 \times 5 = 420$ configurations. Because each configuration was run once under GNU time and once under Hyperfine, this produced 840 successful profiler executions for the main timing-and-memory sweep. All matrices had shape 512×128. Each SVD run used a Jacobi sweep budget of 100 and tolerance 10^{-12}. The benchmark instance was executed sequentially at the job level, so different configurations did not run concurrently. Each configuration was measured once with GNU time for process resource usage and with Hyperfine using 2 warmup runs followed by 5 measured runs for steady-state wall-clock timing. A focused profiler run used representative slices rather than the full factorial space. These slices selected representative layouts, kernels, and matrix families for Valgrind Callgrind, Valgrind Cachegrind, Heaptrack, Massif, DHAT, and mpiP. The profiler run contains 81 successful rows: 24 instruction/cache rows, 45 heap rows, and 12 MPI rows. This separation keeps the expensive profilers from distorting the primary wall-clock comparison while still providing mechanism-level evidence for the main effects.

Table I summarizes the hardware and software platform. The machine exposes 20 logical CPUs from a 12th-generation Intel Core i9-12900H under WSL2. This matters for two reasons. First, thread-level experiments can use multiple logical CPUs inside one benchmark process, but the experiment did not bind threads to cores. Second, Linux perf hardware counters were not available for the WSL2 kernel used in the run, so this evaluation reports elapsed time, user time, system time, maximum resident set size, sweep count, and numerical metrics rather than native hardware counter data. Thus the platform is adequate for controlled end-to-end comparison of this implementation, but conclusions about microarchitectural events such as cache misses should be treated as provisional.

B. Correctness and Measurement Integrity

Table III aggregates the main factorial sweep and the focused profiler run. It combines the Hyperfine status, GNU time status, convergence flags, sweep-budget flags, reconstruction errors, column-correlation residuals, timing coefficients of variation, RSS values, and focused-profiler pass count into one measurement-integrity check.

The first sanity check is that performance differences are not caused by failed SVD runs. Table III shows that all 420 configurations passed the main measurement tools, all reported convergence, and no configuration exhausted the 100-sweep budget. The largest observed relative L_2 reconstruction error was 8.25×10^{-14}, and the largest posterior column-correlation residual was below the requested 10^{-12} tolerance. Hyperfine variability was also modest: the median coefficient of variation was 2.15%, with a 95th percentile of 6.28%. The largest maximum resident set size was 15.1 MiB, so memory footprint differences exist but do not dominate the runtime conclusions.

Table IV separates convergence behavior from timing. The dynamic advanced schedule has the lowest average sweep count, 9.57, while the phase-scheduled mpi-friendly and pthreads kernels average 10.57 sweeps and reach the largest observed sweep count of 27. This sweep difference is real but not a correctness problem: every kernel converges in all 84 configurations, and no kernel produces a tolerance violation. The largest reconstruction error for advanced, 8.25×10^{-14}, is still at ordinary double-precision residual scale for this experiment.

Table V gives the workload-level numerical picture. Degenerate matrices—zero, identity, and diagonal—terminate in one sweep with exactly zero reconstruction error and zero column-correlation residual. Random dense families cluster around 9–10 sweeps and relative L_2 errors below 3.1×10^{-14}. The structured hard cases require more sweeps: low-rank averages 17.8, Hilbert averages 21.4, and ill-conditioned averages 24.4 sweeps. The ill-conditioned family is the numerical worst case, reaching 27 sweeps and a maximum relative L_2 error of 8.25×10^{-14}, but it still satisfies the posterior orthogonality tolerance with zero violation counts.

The important methodological point is that the experiment is measuring a valid design-space comparison for this problem size. It is not measuring failure handling, nonconvergence behavior, or distributed SVD scalability.

C. Layout Effect

Table VI gives the clearest result in the study: layout dominates local Jacobi performance. Column-major storage is the best overall layout, with a geometric-mean speedup of 1.49× against the row-major/naive per-workload baseline. This is exactly what the cost model predicts. One-sided Jacobi repeatedly streams down columns to compute norms, inner products, and two-column rotations. With column-major storage, those streams are unit-stride. With row-major and strided-row-major storage, the same logical operation advances through memory with a large stride. The two row-oriented affine layouts are therefore nearly tied.

Jagged row-major is surprisingly competitive: it is slower than column-major but slightly better than compact row-major in the aggregate. This does not mean pointer chasing is intrinsically good. Rather, at this small dense matrix size, row allocation overhead and handle indirection are not as destructive as the expensive coordinate transforms in blocked and Morton layouts. Blocked row-major roughly doubles the mean wall time relative to row-major because the Jacobi access pattern

TABLE I
Experimental Platform

Component	Configuration
CPU	12th Gen Intel Core i9-12900H
Logical CPUs	20
Cores / threads	10 cores, 2 threads per core
Caches	L1d 480 KiB, L1i 320 KiB, L2 12.5 MiB, L3 24 MiB
Environment	WSL2, Linux 6.6.87.2-microsoft-standard-WSL2
Compiler	GCC 13.3.0, C++20
CMake	3.28.3, release build preset
Python	3.11.9 benchmark driver
Timing tools	Hyperfine 1.18.0, GNU time
Benchmark mode	Sequential jobs, 2 warmups, 5 measured runs
Perf counters	Not reported; unavailable under the WSL2 kernel

TABLE II
Focused Profiler Slice Coverage

Profiler instance	Jobs	Rows	Tools
cache	12	24	Cachegrind, Callgrind
heap	15	45	Heaptrack, Massif, DHAT
mpiP	12	12	mpiP
Total	39	81	six profiler tools

TABLE III
Completeness and Numerical Integrity from Factorial and Profiler Runs

Metric	Value
Main factor combinations	420
Main profiler executions	840
Hyperfine passed	420
GNU time passed	420
Focused profiler rows passed	81
Reported convergence	420
Sweep-budget exhaustion	0
Maximum relative L_2 error	8.25×10^{-14}
Maximum column correlation	1.00×10^{-12}
Hyperfine CV median	2.15%
Hyperfine CV 95th percentile	6.28%
Maximum RSS	15.1 MiB

TABLE IV
Convergence and Numerical Accuracy by Kernel

Kernel	Converged	Mean sweeps	Max sweeps	Max rel. L_2	Violations
naive	84/84	10.07	23	2.88×10^{-14}	0
simd	84/84	10.07	23	2.88×10^{-14}	0
mpi-friendly	84/84	10.57	27	3.50×10^{-14}	0
advanced	84/84	9.57	23	8.25×10^{-14}	0
pthreads	84/84	10.57	27	3.50×10^{-14}	0

does not exploit the two-dimensional tile reuse that the layout tries to preserve.

Morton order is the pathological case. Its mean runtime is 4.67 s and its geometric-mean speed is only 0.046× the baseline. The sweep counts are identical across layouts, so this is not a convergence effect. The slowdown comes from the physical address map: bit interleaving and power-of-two embedding are paid on every scalar access, while this column-oriented kernel receives little locality benefit in return. The lesson is sharp: a layout that is elegant for two-dimensional spatial locality can be the wrong abstraction for a column-streaming numerical kernel.

D. Scheduling and Execution Effect

Kernel differences are more subtle than layout differences. Table VII averages each kernel across all layouts and workloads. The dynamic advanced schedule reduces the average sweep count from 10.07 to 9.57, but it is slower overall because rescoring and sorting all column pairs costs more than the saved rotations at $n = 128$. This is a useful negative result: choosing the largest correlations first is algorithmically plausible, but in this implementation it does not pay for its own scheduling overhead.

The mpi-friendly phase schedule takes slightly more sweeps on average. Its purpose is not to be a faster serial order, but to expose independent matchings that can later be mapped to ranks or threads. In this local experiment, that structural benefit appears as overhead rather than scaling. The pthreads kernel has the lowest mean wall time, 0.488 s, but it also consumes substantially more user and system CPU time. It is exploiting within-process parallelism, not doing less work. Therefore it should be read as a throughput-oriented execution path, while the serial kernels remain cleaner comparisons of layout and schedule.

The simd path is nearly identical to naive in aggregate. This matches the implementation described in Section III: SIMD is currently a dispatch boundary with a scalar fallback rather than a fully vectorized Jacobi microkernel. Its best cases occur when the layout already provides cheap column access, especially column-major storage.

E. Workload Effect

The workload table separates two phenomena: convergence difficulty and fixed overhead. Degenerate families such as zero, identity, and diagonal matrices converge in one sweep, and their best wall-clock times are around 17–20 ms. For these inputs, remaining runtime is mostly program, layout, and output overhead. This is why morton/advanced is still the slowest combination even when the numerical work is trivial: it pays the address-map and

TABLE V
Numerical Correctness and Convergence by Matrix Family

Matrix family	Converged	Mean sweeps	Max sweeps	Max rel. L_2	Max column corr.
zero	30/30	1.0	1	0.00	0.00
identity	30/30	1.0	1	0.00	0.00
diagonal	30/30	1.0	1	0.00	0.00
cauchy	30/30	8.6	10	3.10×10^{-14}	1.00×10^{-12}
sparse	30/30	8.6	9	2.15×10^{-14}	1.00×10^{-12}
rademacher	30/30	9.2	10	2.43×10^{-14}	1.00×10^{-12}
normal	30/30	9.6	10	2.48×10^{-14}	1.00×10^{-12}
integer	30/30	9.6	10	2.36×10^{-14}	9.99×10^{-13}
lognormal	30/30	9.6	10	2.87×10^{-14}	9.99×10^{-13}
uniform	30/30	9.6	10	2.34×10^{-14}	9.99×10^{-13}
banded	30/30	11.0	12	2.62×10^{-14}	1.00×10^{-12}
low-rank	30/30	17.8	23	4.38×10^{-14}	9.99×10^{-13}
hilbert	30/30	21.4	24	2.34×10^{-14}	1.00×10^{-12}
ill-conditioned	30/30	24.4	27	8.25×10^{-14}	1.00×10^{-12}

TABLE VI
Aggregate Layout Effect Across All Kernels and Workloads

Layout	Mean s	Median s	GM speedup	RSS MiB	Sweeps
column-major	0.126	0.093	1.490	10.7	10.17
jagged-row-major	0.200	0.156	1.022	11.2	10.17
row-major	0.233	0.204	0.861	10.7	10.17
strided-row-major	0.234	0.204	0.856	10.6	10.17
blocked-row-major	0.479	0.444	0.377	10.7	10.17
morton	4.669	4.797	0.046	14.3	10.17

TABLE VII
Aggregate Kernel Effect Across All Layouts and Workloads

Kernel	Mean s	Median s	Sweeps	User s	Sys s	RSS MiB
naive	0.954	0.199	10.07	0.97	0.00	11.3
simd	0.971	0.192	10.07	0.98	0.00	11.4
mpi-friendly	1.020	0.203	10.57	1.03	0.00	11.4
advanced	1.518	0.229	9.57	1.54	0.00	11.3
pthreads	0.488	0.344	10.57	2.08	0.40	11.4

scheduling costs before it has any meaningful rotations to perform.

Random dense families such as normal, uniform, integer, lognormal, and Rademacher require about 9–10 sweeps on average. Their best configurations are consistently column-major combined with a simple serial or lightweight phase schedule. Structured hard cases are different. Low-rank, Hilbert, and ill-conditioned matrices require 17.8, 21.4, and 24.4 sweeps on average, respectively. These workloads amplify both numerical iteration cost and any per-access layout penalty, which is why the worst Morton cases reach 16–18 s.

The best combination per workload almost always uses column-major storage. The only exceptions are the one-sweep degenerate matrices, where jagged row-major wins by a few milliseconds. That exception should not be overinterpreted: it is a fixed-overhead result, not evidence that jagged storage is better for a real column-oriented Jacobi workload.

F. Profiler Slice Evidence

The focused profiler slices support the timing interpretation rather than replacing it. Table IX shows that the largest instruction counts are concentrated in Morton jobs, especially normal/morton/advanced. Cachegrind records 1.10×10^{11} instruction references and 5.69×10^{8} branch mispredictions for that job, while normal/morton/naive is also far above the row-major cases. This confirms that Morton is not merely suffering from noisy wall-clock timing; it expands the executed instruction stream through expensive coordinate evaluation and unfavorable control flow.

The heap profilers show a related but smaller effect. Morton jobs peak near 9.0 MB in Massif, whereas representative non-Morton jobs peak around 3.7 MB. DHAT also places normal/morton/advanced at the top of total allocation volume. The memory footprint is not large enough to explain the full slowdown by itself, but it is consistent with the claim that Morton adds structural overhead beyond arithmetic.

The mpiP slice should be read even more narrowly. With two ranks, MPI time accounts for roughly 48–50% of the selected runs; with four ranks, it rises to roughly 71–75%. Since the current implementation does not distribute Jacobi rotations across ranks, these percentages measure MPI scope, launch, barrier, and output-ownership overhead around a local phase-scheduled kernel. They are therefore evidence against claiming distributed scalability from this code path.

G. Failure Modes and Limits

Table X lists the largest slowdowns relative to the row-major/naive baseline within each matrix family. The concentration is absolute: all worst cases use Morton layout, and the largest six use either advanced or mpi-friendly. This exposes a bad interaction rather than an isolated bad component. Morton makes every scalar access expensive; dynamic scheduling increases the number of full-pair scans; phase scheduling changes the traversal shape without removing the address-map cost. These effects multiply.

The main limitation is external validity. The experiment fixes one shape, one tolerance, one sweep budget, one machine, and a WSL2 environment. The results are therefore strongest as an internal comparison of

TABLE VIII
Best and Worst Configuration by Matrix Family

Matrix family	Sweeps	Best layout/kernel	Best s	Worst layout/kernel	Worst s
zero	1.0	jagged-row-major/simd	0.017	morton/advanced	0.832
identity	1.0	jagged-row-major/naive	0.020	morton/advanced	0.955
diagonal	1.0	jagged-row-major/naive	0.020	morton/advanced	0.869
cauchy	8.6	column-major/mpi-friendly	0.080	morton/advanced	5.445
sparse	8.6	column-major/simd	0.081	morton/advanced	5.627
rademacher	9.2	column-major/mpi-friendly	0.082	morton/advanced	6.260
normal	9.6	column-major/naive	0.084	morton/advanced	6.407
integer	9.6	column-major/naive	0.092	morton/advanced	6.327
lognormal	9.6	column-major/simd	0.086	morton/advanced	6.334
uniform	9.6	column-major/naive	0.091	morton/advanced	6.611
banded	11.0	column-major/mpi-friendly	0.091	morton/advanced	6.988
low-rank	17.8	column-major/simd	0.119	morton/advanced	18.312
hilbert	21.4	column-major/simd	0.132	morton/advanced	18.243
ill-conditioned	24.4	column-major/naive	0.154	morton/advanced	16.619

TABLE IX
Representative Findings from Focused Profiler Slices

Profiler	Metric	Highest observed job	Value	Interpretation
Cachegrind	Instruction refs	normal/morton/advanced	1.10×10^{11}	Morton address map dominates instruction stream
Cachegrind	Branch mispredicts	normal/morton/advanced	5.69×10^{8}	Coordinate/control overhead is visible
Callgrind	Instruction refs	normal/morton/advanced	1.10×10^{11}	Confirms Cachegrind ranking
Massif	Peak heap bytes	normal/morton/pthreads	9.03×10^{6}	Morton storage has larger live footprint
DHAT	Total allocated bytes	normal/morton/advanced	2.16×10^{7}	Dynamic Morton path allocates most
mpiP	MPI time share	normal/morton/mpi-friendly, np4	74.95%	MPI wrapper overhead dominates local kernel run

TABLE X
Largest Slowdowns Relative to Row-Major/Naive Within Each Workload

Matrix	Layout	Kernel	Slowdown
low-rank	morton	advanced	63.16×
hilbert	morton	advanced	58.61×
ill-conditioned	morton	advanced	46.60×
zero	morton	advanced	42.22×
identity	morton	advanced	39.68×
diagonal	morton	advanced	38.23×
hilbert	morton	mpi-friendly	36.66×
banded	morton	advanced	32.31×
rademacher	morton	advanced	32.10×
integer	morton	advanced	32.05×

the implemented design choices. They should not be generalized to all SVD sizes, to a native Linux HPC node, or to a distributed-memory SVD solver without additional experiments. Within that scope, however, the evidence is consistent: for the present one-sided Jacobi implementation, simple column-contiguous storage is the dominant design decision, while more elaborate scheduling and layout transformations need a much stronger workload-specific reason to justify their overhead.

V. Discussion

The evaluation should be read as a decomposition of cost, not as a raw leaderboard of 420 runs. The factorial space separates three mechanisms that often collapse into one wall-clock number in small parallel numerical programs. Layout changes the memory trajectory of the same mathematical operation; scheduling changes both convergence and observation overhead; execution policy changes elapsed time, CPU work, and eventually communication. The main empirical claim is therefore local but strong: before rank-level distribution is introduced, the bottleneck is already shaped by data movement and access structure. This is consistent with the long line of memory-hierarchy and communication-avoiding work showing that arithmetic count alone is an insufficient performance model for numerical linear algebra [4], [5], [12], [13], [26].

A. Layout Is Part of the Algorithm

One-sided Jacobi SVD is a column-major algorithm in the algorithmic sense, even if the container is not stored in column-major order. Each pivot step chooses two columns g_i and g_j, computes their norms and inner product, and then applies a two-column rotation. Thus the hot path repeatedly walks down columns of G and also updates columns of V. The relevant architectural question is not whether the matrix is two-dimensional, but whether this column walk is a unit-stride stream.

A simple cache-line model captures the layout effect. Let B be the number of doubles per cache line and let δ_L be the physical stride, in doubles, between consecutive logical entries of one column under layout L. For a column segment of length k, approximate the number of distinct cache lines touched by

$$\text{lines}(k, \delta_L) = \Theta\left(\min\left\{k, \left\lceil \frac{k\delta_L}{B} \right\rceil\right\}\right). \tag{75}$$

For column-major storage, $\delta_L = 1$, so a column scan costs $\Theta(k/B)$ cache lines. For compact row-major storage in this benchmark, $\delta_L = n = 128$. Since a typical 64-byte cache line holds $B = 8$ doubles, the model reduces the

row-major column scan to $\Theta(k)$ useful cache-line touches: almost every loaded line contributes only one scalar to the current column operation. Strided row-major has the same structure with $\delta_L = p$, which explains why row-major and strided-row-major are nearly tied in Table VI.

For one Jacobi pair, a first-order memory cost can therefore be written as

$$Q_L^{\text{pair}} = a_G \text{ lines}(m, \delta_L) + a_V \text{ lines}(n, \delta_V) + c_L^{\text{addr}}(m+n), \tag{76}$$

where the constants a_G and a_V account for reading and writing the working matrix and the right-singular-vector accumulator, and c_L^{addr} is the per-access address-evaluation cost. Equation (76) is deliberately simple, but it explains the observed ordering. Column-major reduces cache-line traffic for the dominant m-length streams. Row-major and pitched row-major lose spatial locality. Blocked row-major helps only if the algorithm reuses two-dimensional tiles; this scalar column-pair kernel does not. Morton order has no stable one-dimensional column stride and pays a larger c_L^{addr} through recursive coordinate evaluation. Its intended advantage is cache-oblivious two-dimensional locality [16], but the measured kernel asks for long one-dimensional streams instead.

Putting the benchmark constants into the model turns this into a quantitative explanation. With $m = 512$ and a typical cache line holding $B = 8$ doubles, one contiguous G-column scan touches about 64 cache lines, whereas a row-major column traversal can touch roughly 512 lines for the same logical vector. The observed layout means, 0.126 s for column-major and 0.233 s for row-major, are not expected to differ by a full factor of eight because arithmetic, updates to V, fixed loop overhead, partial cache reuse, and measurement noise dilute raw line traffic. The sign and scale still follow the model. Morton is worse than a pure stride argument would predict because it increases both the irregular line function and the c_L^{addr} term; the profiler's instruction and branch counts are therefore part of the same explanation. The jagged layout is competitive at this size because the row handles and fragments remain cache-resident, but Equation (76) predicts that this advantage is fragile as the allocation footprint grows.

This is why all layouts have the same average sweep count but sharply different runtimes. The Jacobi iteration sees the same mathematical pairs; the memory hierarchy sees different streams. The profiler slice supports the model: normal/morton/advanced has the largest instruction and branch-misprediction counts, while the column-major cases stay close to the scalar cyclic baseline. The practical direction is also clear from high-performance dense linear algebra: successful implementations often pack or block data around the kernel that will consume it, rather than exposing arbitrary coordinate maps to the innermost loop [15], [25]. For this implementation, the natural packed object is a column-contiguous panel.

B. Scheduling Needs a Cost Model

The dynamic schedule should be evaluated by total cost per sweep, not by sweep count alone. Let

$$P_n = \binom{n}{2} \tag{77}$$

be the number of column pairs. For a schedule π, write the cost of sweep s as

$$T_{\pi,s} = C_{\pi,s}^{\text{sched}} + P_n C_L^{\text{test}} + R_{\pi,s} C_L^{\text{rot}}, \tag{78}$$

where C_L^{test} is the cost of testing one pair, C_L^{rot} is the additional cost of applying a rotation, and $R_{\pi,s}$ is the number of active rotations in that sweep. For a cyclic schedule, $C_{\pi,s}^{\text{sched}}$ is essentially loop overhead. For the dynamic advanced schedule,

$$C_{\text{dyn},s}^{\text{sched}} = P_n C_L^{\text{score}} + \Theta(P_n \log P_n) C^{\text{sort}}, \tag{79}$$

because every pair is rescored and the pair list is sorted before the sweep begins.

Dynamic ordering is profitable only if the saved Jacobi work exceeds this observation cost:

$$\Delta W_{\text{saved}} > \sum_s C_{\text{dyn},s}^{\text{sched}}. \tag{80}$$

Here a simple approximation for the saved work is

$$\Delta W_{\text{saved}} = \sum_s (R_{\text{cyc},s} - R_{\text{dyn},s}) C_L^{\text{rot}} + (S_{\text{cyc}} - S_{\text{dyn}}) P_n C_L^{\text{test}}. \tag{81}$$

At $n = 128$, $P_n = 8128$. The aggregate sweep reduction in Table IV is only $10.07 - 9.57 = 0.50$ sweeps, roughly 4064 pair visits. In exchange, the dynamic schedule performs about $9.57 \times 8128 \approx 7.8 \times 10^4$ pair scores and sorts a list of 8128 pairs in each dynamic sweep. Equation (80) therefore predicts the result in Table VII: the schedule can reduce sweeps and still lose wall-clock time.

The measured means show the inequality failing in exactly this way. The advanced kernel averages 1.518 s, while the cyclic naive kernel averages 0.954 s. A half-sweep saving removes about 4064 pair visits, but the scheduler collects about 7.8×10^4 pair scores, or roughly nineteen score observations per saved visit, before sorting is counted. The saved-rotation term would have to be very large to compensate for that observation work. In some structured cases it is not even positive: for Hilbert matrices in Table XII, advanced takes 23 sweeps while naive takes 18. There the model predicts both more Jacobi visits and more scheduling overhead, which is exactly why a convergence heuristic can be mathematically plausible but experimentally slower.

This does not contradict prior block-Jacobi work where dynamic ordering and variable blocking can reduce iteration steps enough to matter [27]. The distinction is granularity. Block-level dynamic ordering observes fewer, heavier subproblems, so one scheduling decision can amortize over more floating-point work and more data reuse. The current advanced path is element-wise and global: it pays to observe all pair correlations even when the next sweep will quickly make parts of that order stale. A better

next schedule would preserve the phase structure but lower the observation frequency, for example by using block-level scores, thresholded rescans, or stale priority information that is refreshed only after several sweeps.

C. Parallelism Does Not Mean Less Work

Parallel execution needs a work, span, and communication model. For a phase schedule with $q = n - 1$ phases when n is even, each phase contains at most $\lfloor n/2 \rfloor$ independent column pairs. A shared-memory execution can be approximated by

$$T_{\text{thr}} \approx \sum_{s=1}^{S} \sum_{\phi=1}^{q} \left(\frac{W_{s,\phi}^{\text{pair}}}{p_{\text{eff}}} + \alpha_{\text{thr}} + \beta_{\text{coh}} M_{s,\phi} \right), \tag{82}$$

where $W_{s,\phi}^{\text{pair}}$ is the useful pair work in one phase, p_{eff} is the effective parallelism after imbalance and memory bandwidth limits, α_{thr} is scheduling or synchronization latency, and $M_{s,\phi}$ represents coherence and memory-traffic pressure. The corresponding total CPU work is not T_{thr}; it is closer to

$$W_{\text{thr}} = \sum_{s,\phi} W_{s,\phi}^{\text{pair}} + W^{\text{sched}} + W^{\text{sync}} + W^{\text{coh}}. \tag{83}$$

This is the model behind the Pthreads result. The Pthreads kernel has the lowest mean elapsed time, but it also consumes much more user and system CPU time. It reduces span by spending more cores and more runtime-system work; it does not make the Jacobi computation intrinsically smaller.

Substituting the aggregate measurements gives the model an empirical scale. Pthreads reduce mean elapsed time from 0.954 s to 0.488 s, only a 1.96× speedup, while mean CPU work rises from about 0.97 s to $2.08 + 0.40 = 2.48$ s. In Equation (82), the useful span term shrinks, but W^{sched}, W^{sync}, and W^{coh} become visible. The same model also predicts why the thread result depends on layout. When column-major storage already makes each pair cheap, thread overhead can dominate the saved span; when Morton order makes each pair expensive, the larger $W_{s,\phi}^{\text{pair}}$ gives the runtime more work over which to amortize scheduling and synchronization. The raw normal-matrix cases make the contrast explicit: col/pth is slower than col/naive (0.2288 s versus 0.0836 s), while morton/pth is much faster than morton/naive (1.2238 s versus 4.7788 s). Pthreads are therefore not a universal work reducer, but a way to trade extra CPU work for lower elapsed time when the local pair work is large enough.

The same distinction becomes sharper for distributed memory. A rank-level Jacobi implementation would have a cost of the form

$$T_{\text{dist}} \approx \frac{W_{\text{local}}}{p_{\text{eff}}} + \sum_{s=1}^{S} \sum_{\phi=1}^{q} \left(\alpha_{\text{net}} h_{s,\phi} + \beta_{\text{net}} b_{s,\phi} + \gamma_{\text{sync}} \right), \tag{84}$$

where $h_{s,\phi}$ is the number of messages, $b_{s,\phi}$ is the number of bytes moved, and α_{net}, β_{net}, and γ_{sync} model latency, inverse bandwidth, and synchronization. This is the standard reason communication-avoiding linear algebra treats data movement as a first-class cost [4], [5]. For SVD at extreme scale, the bottleneck similarly shifts among local kernels, memory movement, and synchronization [25].

The present MPI-friendly path does not yet instantiate the useful part of Equation (84). It exposes independent phases, but it does not exchange column blocks and perform rotations across ranks. Thus the mpiP percentages measure wrapper, barrier, launch, and ownership overhead around local work. They are valuable precisely because they prevent an overclaim: the code has an MPI-shaped dependency structure, not an MPI-scaled SVD.

The model also indicates what a real distributed experiment would have to explain. If a block pair has block width b, one phase must make the active G panels and the corresponding V panels available to the ranks that apply the rotations. Without replication, this is on the order of $\Theta(mb)$ doubles for G and $\Theta(nb)$ doubles for V per active block pair, plus latency for the exchange and synchronization at phase boundaries. The useful local block work is roughly $\Theta((m+n)b^2)$, so communication is amortized only when b is large enough, or when several local rotations reuse the same panels before they move again. This is the concrete meaning of "parallelism does not mean less work" for the MPI path: at scale, the question is not only whether pairs are independent, but whether the independence buys enough local work to pay for panel motion.

D. Workload Structure Explains Sweep Count

One-sided Jacobi SVD diagonalizes the Gram matrix $C = G^T G$ by right rotations. For nonzero columns, let $D = \text{diag}(\|g_1\|_2, \ldots, \|g_n\|_2)$ and define the normalized Gram matrix

$$H = D^{-1} C D^{-1}. \tag{85}$$

Then $H_{ii} = 1$ and H_{ij} is the normalized column correlation. A useful scalar measure of remaining pairwise nonorthogonality is

$$\eta(G) = \|H - I\|_F^2 = 2 \sum_{i<j} \rho_{ij}^2. \tag{86}$$

A Jacobi rotation annihilates one current pair correlation, but it also changes the other correlations involving the two rotated columns. Therefore the number of sweeps is controlled not only by $\eta(G^{(0)})$, but also by how quickly pair rotations stop recontaminating other pairs. As a first-order convergence picture,

$$S \approx \frac{\log(\eta(G^{(0)})/\varepsilon^2)}{-\log \mu(A, \pi)} \tag{87}$$

until the iteration reaches its asymptotic regime, where $\mu(A, \pi)$ is an effective contraction factor determined by matrix structure and pivot ordering. This is not a theorem for the present implementation; it is a diagnostic model for interpreting the sweep table.

The formula separates two mechanisms that the raw sweep count alone hides. The numerator says how much normalized off-diagonal energy must be removed at the start; the denominator says how much of the remaining energy a sweep removes. Thus a matrix can be hard because $\eta(G^{(0)})$ is large, because $\mu(A, \pi)$ is close to one, or because both are true. The degenerate families satisfy $\eta(G^{(0)}) = 0$, so they require only one confirming sweep. For random dense matrices with $m = 512$, typical normalized pair correlations scale like $O(m^{-1/2}) \approx 0.044$: there are many nonzero correlations, but they are diffuse rather than concentrated in a low-dimensional column geometry. That gives the observed 9–10 sweep regime. Low-rank inputs raise the effective numerator by putting many columns in the same small subspace, and rotations of one pair can recontaminate other pairs in that subspace. Hilbert and ill-conditioned inputs mainly damage the denominator: near-dependence and small spectral gaps make each sweep contract the remaining off-diagonal energy slowly. This is why the workload table orders the hard cases as low-rank, Hilbert, and ill-conditioned at 17.8, 21.4, and 24.4 sweeps rather than treating them as merely larger versions of the random case.

The correlation-graph view adds one more distinction. Banded and sparse matrices may have fewer strongly coupled pairs than dense matrices, but their nonzero pattern can create local clusters of column correlations. A sweep that fixes one edge of such a cluster can perturb adjacent edges, so the relevant structure is not only the number of nonzeros but the geometry of the induced correlation graph. For Hilbert and ill-conditioned matrices, perturbation theory for the SVD makes the gap-dependence explicit [17]–[19], while numerical analysis texts use Hilbert-type matrices exactly because they expose conditioning limits [21]. The sweep table is therefore a structural measurement: it reports how column geometry, spectral gaps, and pivot order interact, not merely how many flops were executed.

This also clarifies what the accuracy table does and does not prove. Small reconstruction error means the factorization reconstructs A well; it does not fully characterize the stability of singular vectors or invariant subspaces when singular values are clustered. A stronger numerical study would compare singular values to a trusted LAPACK driver, measure subspace angles for known spectral clusters, and report how $\eta(G)$ decays sweep by sweep.

E. Threats to Validity and Next Steps

The models above are intentionally first-order. The experiment fixes one shape, one tolerance, one sweep budget, one machine, one compiler configuration, and one WSL2 environment. The direction of some effects is likely robust: a column-streaming kernel should generally prefer column-contiguous storage, and a global dynamic schedule must pay for its observations. The magnitudes are not universal. Larger m could strengthen the locality advantage of column panels; larger n could make dynamic sorting and phase synchronization more expensive; different aspect ratios could shift the balance between updating G and updating V.

The implementation boundaries also matter. The Morton and blocked layouts are evaluated as coordinate maps under a scalar one-sided Jacobi kernel, not as complete recursive or blocked SVD algorithms. The SIMD path is a dispatch boundary with scalar fallback, so it cannot answer the question of a fully vectorized column microkernel. The MPI path prepares a phase dependency graph, but it does not yet implement rank-level rotations.

The next implementation step should follow the models rather than only the rankings. First, make the local unit of computation a column-contiguous panel, either by storing that way or by explicit packing. Second, lift the phase schedule from column pairs to column blocks so that one scheduling decision amortizes over more arithmetic and memory reuse. Third, implement actual rank-level block exchanges and evaluate Equation (84): bytes moved, message count, synchronization, local panel performance, and convergence should be reported separately. Only then would the MPI experiment answer the real scalability question.

VI. Conclusion

This paper studied one-sided Jacobi SVD as an implementation design space rather than as a single fixed routine. Across 420 factorial configurations and 81 focused profiler rows, the dominant empirical lesson is that local data layout matters more than the more elaborate scheduling choices tested here. Column-major storage gives the best aggregate performance because the Jacobi kernel is fundamentally a column-streaming computation. Morton order gives the opposite result: its recursive address map is elegant for two-dimensional locality, but it is a poor fit for repeated scalar access along columns.

The kernel results refine that lesson. Dynamic correlation ordering reduces the average sweep count, but it does not repay the cost of rescoring and sorting all column pairs at this problem size. The phase schedule is still worth keeping, not because it is the fastest serial order, but because it exposes independent matchings that are necessary for thread-level and future rank-level execution. The Pthreads path demonstrates useful local elapsed-time reduction, while also showing that lower wall time can come from higher total CPU work rather than from a cheaper algorithm.

The present implementation should therefore not be read as evidence of distributed SVD scalability. Its MPI-friendly kernel prepares a dependency structure, but it does not yet distribute Jacobi rotations across ranks. The natural next step is to turn the phase schedule into a real rank-level column-block implementation: pack or store local panels in column-contiguous form, assign independent block pairs to ranks, batch communication at phase boundaries, and then measure whether communication and synchronization can be amortized over a strong local Jacobi kernel.

References

[1] G. H. Golub and C. Reinsch, "Singular value decomposition and least squares solutions," Numerische Mathematik, vol. 14, no. 5, pp. 403–420, 1970.

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

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK Users' Guide, 3rd ed. Philadelphia, PA, USA: SIAM, 1999.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," SIAM Journal on Matrix Analysis and Applications, vol. 32, no. 3, pp. 866–901, 2011.

[5] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz, "Communication lower bounds and optimal algorithms for numerical linear algebra," Acta Numerica, vol. 23, pp. 1–155, 2014.

[6] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," SIAM Review, vol. 53, no. 2, pp. 217–288, 2011.

[7] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," Journal of the Society for Industrial and Applied Mathematics, vol. 6, no. 1, pp. 51–90, 1958.

[8] J. Demmel and K. Veselić, "Jacobi's method is more accurate than QR," SIAM Journal on Matrix Analysis and Applications, vol. 13, no. 4, pp. 1204–1245, 1992.

[9] Z. Drmač and K. Veselić, "New fast and accurate jacobi SVD algorithm. I," SIAM Journal on Matrix Analysis and Applications, vol. 29, no. 4, pp. 1322–1342, 2008.

[10] P. P. M. de Rijk, "A one-sided jacobi algorithm for computing the singular value decomposition on a vector computer," SIAM Journal on Scientific and Statistical Computing, vol. 10, no. 2, pp. 359–371, 1989.

[11] P. J. Eberlein, "On one-sided jacobi methods for parallel computation," SIAM Journal on Algebraic and Discrete Methods, vol. 8, no. 4, pp. 790–796, 1987.

[12] K. Gallivan, W. Jalby, U. Meier, and A. H. Sameh, "Impact of hierarchical memory systems on linear algebra algorithm design," The International Journal of Supercomputing Applications, vol. 2, no. 1, pp. 12–48, 1988.

[13] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63–74.

[14] U. Drepper, "What every programmer should know about memory," Red Hat, Inc., Tech. Rep., 2007. [Online]. Available: https://www.akkadia.org/drepper/cpumemory.pdf

[15] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," ACM Transactions on Mathematical Software, vol. 34, no. 3, pp. 12:1–12:25, 2008.

[16] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in Proceedings of the 40th Annual Symposium on Foundations of Computer Science, 1999, pp. 285–297.

[17] P.-Å. Wedin, "Perturbation bounds in connection with singular value decomposition," BIT Numerical Mathematics, vol. 12, no. 1, pp. 99–111, 1972.

[18] G. W. Stewart and J. guang Sun, Matrix Perturbation Theory. Boston, MA, USA: Academic Press, 1990.

[19] G. W. Stewart, "Perturbation theory for the singular value decomposition," in SVD and Signal Processing, II: Algorithms, Analysis and Applications, R. J. Vaccaro, Ed. Amsterdam, The Netherlands: Elsevier, 1991.

[20] A. Edelman, "Eigenvalues and condition numbers of random matrices," SIAM Journal on Matrix Analysis and Applications, vol. 9, no. 4, pp. 543–560, 1988.

[21] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[22] P. C. Hansen, Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion. Philadelphia, PA, USA: SIAM, 1998.

[23] V. Novaković, "Vectorization of a thread-parallel jacobi singular value decomposition method," SIAM Journal on Scientific Computing, vol. 45, no. 3, pp. C143–C165, 2023.

[24] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed. MIT Press, 1999.

[25] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale," SIAM Review, vol. 60, no. 4, pp. 808–865, 2018.

[26] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, 1981, pp. 326–333.

[27] S. Kudo, Y. Yamamoto, M. Bečka, and M. Vajteršic, "Performance analysis and optimization of the parallel one-sided block jacobi SVD algorithm with dynamic ordering and variable blocking," Concurrency and Computation: Practice and Experience, vol. 29, no. 9–10, p. e4059, 2017.

Appendix A
Supplementary Tables

This appendix records the compact evaluation design space and the configuration-level raw results used by the experiments. The raw-data tables use abbreviated codes to keep the appendix readable: adv denotes the dynamic advanced kernel, mpi the MPI-friendly phase schedule, pth the Pthreads kernel, col column-major layout, row row-major layout, block-row blocked row-major layout, jagged-row jagged row-major layout, and stride-row strided row-major layout.

TABLE XI
Full Factorial Configuration Space

Factor	Values
Matrix family	14 synthetic families
Layout	6 MatrixLike representations
Kernel	5 one-sided Jacobi variants
Problem size	512×128
Sweep budget	100
Tolerance	10^{-12}
Total configurations	420

TABLE XII: Configuration-Level Factorial Raw Results

M	L	K	Mean s	SD s	RSS KiB	Sweeps	Rel. L2	Max corr.
banded	block-row	adv	0.6682	0.033	9368	9	1.887e-14	9.943e-13
banded	block-row	mpi	0.4847	0.011	9372	11	2.618e-14	9.995e-13
banded	block-row	naive	0.5237	0.017	9368	12	2.622e-14	9.984e-13
banded	block-row	pth	0.4038	0.007	9504	11	2.618e-14	9.995e-13
banded	block-row	simd	0.5128	0.008	9372	12	2.622e-14	9.984e-13
banded	col	adv	0.1264	0.003	9368	9	1.887e-14	9.943e-13
banded	col	mpi	0.0908	0.001	9368	11	2.618e-14	9.995e-13
banded	col	naive	0.0922	0.005	9096	12	2.622e-14	9.984e-13
banded	col	pth	0.2469	0.002	9492	11	2.618e-14	9.995e-13
banded	col	simd	0.0969	0.003	9368	12	2.622e-14	9.984e-13
banded	jagged-row	adv	0.1849	0.004	9920	9	1.887e-14	9.943e-13
banded	jagged-row	mpi	0.1618	0.001	9600	11	2.618e-14	9.995e-13
banded	jagged-row	naive	0.1701	0.003	9916	12	2.622e-14	9.984e-13
banded	jagged-row	pth	0.3811	0.004	9916	11	2.618e-14	9.995e-13
banded	jagged-row	simd	0.1706	0.004	9916	12	2.622e-14	9.984e-13
banded	morton	adv	6.9878	0.091	14200	9	1.887e-14	9.943e-13
banded	morton	mpi	5.213	0.219	13960	11	2.618e-14	9.995e-13
banded	morton	naive	5.5605	0.12	13956	12	2.622e-14	9.984e-13
banded	morton	pth	1.3473	0.015	14168	11	2.618e-14	9.995e-13
banded	morton	simd	5.6836	0.072	14172	12	2.622e-14	9.984e-13
banded	row	adv	0.2321	0.01	9372	9	1.887e-14	9.943e-13
banded	row	mpi	0.1914	0.002	9372	11	2.618e-14	9.995e-13
banded	row	naive	0.2163	0.007	9372	12	2.622e-14	9.984e-13
banded	row	pth	0.3336	0.012	9316	11	2.618e-14	9.995e-13
banded	row	simd	0.2081	0.01	9368	12	2.622e-14	9.984e-13
banded	stride-row	adv	0.2286	0.003	9368	9	1.887e-14	9.943e-13
banded	stride-row	mpi	0.2085	0.002	9240	11	2.618e-14	9.995e-13
banded	stride-row	naive	0.2161	0.008	9372	12	2.622e-14	9.984e-13
banded	stride-row	pth	0.3526	0.006	9268	11	2.618e-14	9.995e-13
banded	stride-row	simd	0.2184	0.005	9372	12	2.622e-14	9.984e-13
cauchy	block-row	adv	0.5202	0.009	12412	7	3.097e-14	9.997e-13
cauchy	block-row	mpi	0.3753	0.017	12140	8	2.676e-14	9.998e-13
cauchy	block-row	naive	0.4212	0.009	12416	10	2.348e-14	9.999e-13
cauchy	block-row	pth	0.315	0.018	12468	8	2.676e-14	9.998e-13
cauchy	block-row	simd	0.4309	0.01	12416	10	2.348e-14	9.999e-13
cauchy	col	adv	0.1093	0.003	12416	7	3.097e-14	9.997e-13
cauchy	col	mpi	0.0802	0.001	12412	8	2.676e-14	9.998e-13
cauchy	col	naive	0.0901	0.004	12412	10	2.348e-14	9.999e-13
cauchy	col	pth	0.1942	0.003	12576	8	2.676e-14	9.998e-13
cauchy	col	simd	0.081	0.001	12412	10	2.348e-14	9.999e-13
cauchy	jagged-row	adv	0.147	0.006	12980	7	3.097e-14	9.997e-13
cauchy	jagged-row	mpi	0.1301	0.009	12976	8	2.676e-14	9.998e-13
cauchy	jagged-row	naive	0.1346	0.002	12976	10	2.348e-14	9.999e-13
cauchy	jagged-row	pth	0.2751	0.004	12880	8	2.676e-14	9.998e-13
cauchy	jagged-row	simd	0.1395	0.002	12976	10	2.348e-14	9.999e-13
cauchy	morton	adv	5.4447	0.094	15324	7	3.097e-14	9.997e-13
cauchy	morton	mpi	4.0354	0.058	15296	8	2.676e-14	9.998e-13
cauchy	morton	naive	4.6682	0.111	15336	10	2.348e-14	9.999e-13
cauchy	morton	pth	1.0949	0.018	15268	8	2.676e-14	9.998e-13
cauchy	morton	simd	4.8585	0.087	15292	10	2.348e-14	9.999e-13
cauchy	row	adv	0.1919	0.004	12140	7	3.097e-14	9.997e-13
cauchy	row	mpi	0.1628	0.006	12416	8	2.676e-14	9.998e-13
cauchy	row	naive	0.1805	0.006	12412	10	2.348e-14	9.999e-13
cauchy	row	pth	0.2572	0.011	12320	8	2.676e-14	9.998e-13

Continued on next page

Configuration-Level Factorial Raw Results (continued)

M	L	K	Mean s	SD s	RSS KiB	Sweeps	Rel. L2	Max corr.
cauchy	row	simd	0.1683	0.003	12416	10	2.348e-14	9.999e-13
cauchy	stride-row	adv	0.1906	0.004	12412	7	3.097e-14	9.997e-13
cauchy	stride-row	mpi	0.1935	0.009	12416	8	2.676e-14	9.998e-13
cauchy	stride-row	naive	0.1774	0.007	12416	10	2.348e-14	9.999e-13
cauchy	stride-row	pth	0.3072	0.005	12372	8	2.676e-14	9.998e-13
cauchy	stride-row	simd	0.1838	0.01	12136	10	2.348e-14	9.999e-13
diag	block-row	adv	0.118	0.007	8204	1	0	0
diag	block-row	mpi	0.0941	0.002	8032	1	0	0
diag	block-row	naive	0.0866	0.002	8204	1	0	0
diag	block-row	pth	0.0956	0.003	7812	1	0	0
diag	block-row	simd	0.0967	0.003	8208	1	0	0
diag	col	adv	0.0268	0.001	7776	1	0	0
diag	col	mpi	0.0213	9.523e-04	8100	1	0	0
diag	col	naive	0.0228	8.138e-04	8020	1	0	0
diag	col	pth	0.0384	3.278e-04	8192	1	0	0
diag	col	simd	0.0216	9.657e-04	8024	1	0	0
diag	jagged-row	adv	0.0286	5.886e-04	8596	1	0	0
diag	jagged-row	mpi	0.0208	5.971e-04	8756	1	0	0
diag	jagged-row	naive	0.0203	3.943e-04	8592	1	0	0
diag	jagged-row	pth	0.0382	6.986e-04	8312	1	0	0
diag	jagged-row	simd	0.0208	0.001	8752	1	0	0
diag	morton	adv	0.8686	0.028	14044	1	0	0
diag	morton	mpi	0.6162	0.005	14016	1	0	0
diag	morton	naive	0.6322	0.006	14012	1	0	0
diag	morton	pth	0.4049	0.005	14032	1	0	0
diag	morton	simd	0.6203	0.012	14016	1	0	0
diag	row	adv	0.0312	0.003	8180	1	0	0
diag	row	mpi	0.0257	0.001	8028	1	0	0
diag	row	naive	0.0227	8.928e-04	8048	1	0	0
diag	row	pth	0.0394	0.002	7876	1	0	0
diag	row	simd	0.0225	0.001	8044	1	0	0
diag	stride-row	adv	0.0295	0.001	8040	1	0	0
diag	stride-row	mpi	0.022	0.001	8236	1	0	0
diag	stride-row	naive	0.0216	0.001	8044	1	0	0
diag	stride-row	pth	0.0366	9.496e-04	8024	1	0	0
diag	stride-row	simd	0.0217	5.458e-04	8204	1	0	0
hilbert	block-row	adv	1.6118	0.012	12256	23	2.341e-14	9.987e-13
hilbert	block-row	mpi	0.9877	0.019	12416	24	2.272e-14	9.994e-13
hilbert	block-row	naive	0.7262	0.011	12416	18	2.734e-15	9.995e-13
hilbert	block-row	pth	0.7866	0.014	12360	24	2.272e-14	9.994e-13
hilbert	block-row	simd	0.7305	0.004	12416	18	2.734e-15	9.995e-13
hilbert	col	adv	0.2885	0.005	12412	23	2.341e-14	9.987e-13
hilbert	col	mpi	0.1789	0.003	12412	24	2.272e-14	9.994e-13
hilbert	col	naive	0.1347	0.003	12416	18	2.734e-15	9.995e-13
hilbert	col	pth	0.525	0.008	12304	24	2.272e-14	9.994e-13
hilbert	col	simd	0.1319	0.007	12136	18	2.734e-15	9.995e-13
hilbert	jagged-row	adv	0.5021	0.009	12972	23	2.341e-14	9.987e-13
hilbert	jagged-row	mpi	0.3384	0.006	12976	24	2.272e-14	9.994e-13
hilbert	jagged-row	naive	0.2424	0.002	12976	18	2.734e-15	9.995e-13
hilbert	jagged-row	pth	0.7897	0.02	12884	24	2.272e-14	9.994e-13
hilbert	jagged-row	simd	0.2447	0.006	12820	18	2.734e-15	9.995e-13
hilbert	morton	adv	18.243	0.102	15268	23	2.341e-14	9.987e-13
hilbert	morton	mpi	11.4112	0.115	15428	24	2.272e-14	9.994e-13
hilbert	morton	naive	8.3615	0.036	15240	18	2.734e-15	9.995e-13
hilbert	morton	pth	2.5289	0.052	15452	24	2.272e-14	9.994e-13
hilbert	morton	simd	8.5516	0.117	15456	18	2.734e-15	9.995e-13
hilbert	row	adv	0.6399	0.021	12416	23	2.341e-14	9.987e-13
hilbert	row	mpi	0.4515	0.01	12412	24	2.272e-14	9.994e-13
hilbert	row	naive	0.3113	0.006	12416	18	2.734e-15	9.995e-13
hilbert	row	pth	0.7818	0.035	12340	24	2.272e-14	9.994e-13
hilbert	row	simd	0.3299	0.006	12140	18	2.734e-15	9.995e-13
hilbert	stride-row	adv	0.6087	0.029	12256	23	2.341e-14	9.987e-13
hilbert	stride-row	mpi	0.4485	0.024	12416	24	2.272e-14	9.994e-13
hilbert	stride-row	naive	0.3049	0.012	12416	18	2.734e-15	9.995e-13
hilbert	stride-row	pth	0.7537	0.02	12416	24	2.272e-14	9.994e-13
hilbert	stride-row	simd	0.3486	0.02	12416	18	2.734e-15	9.995e-13
ident	block-row	adv	0.1103	0.003	8032	1	0	0
ident	block-row	mpi	0.0832	0.001	8208	1	0	0
ident	block-row	naive	0.0838	7.347e-04	8024	1	0	0
ident	block-row	pth	0.0856	0.003	8004	1	0	0
ident	block-row	simd	0.0873	0.001	8048	1	0	0
ident	col	adv	0.0265	0.001	8044	1	0	0
ident	col	mpi	0.022	9.806e-04	8152	1	0	0
ident	col	naive	0.0213	8.449e-04	8028	1	0	0
ident	col	pth	0.036	8.110e-04	7832	1	0	0
ident	col	simd	0.0211	4.209e-04	8028	1	0	0
ident	jagged-row	adv	0.0261	9.739e-04	8728	1	0	0
ident	jagged-row	mpi	0.0208	0.001	8596	1	0	0
ident	jagged-row	naive	0.0199	5.723e-04	8592	1	0	0
ident	jagged-row	pth	0.035	0.001	8316	1	0	0
ident	jagged-row	simd	0.02	7.741e-04	8592	1	0	0
ident	morton	adv	0.9553	0.056	14044	1	0	0
ident	morton	mpi	0.6512	0.035	14020	1	0	0
ident	morton	naive	0.6212	0.005	14012	1	0	0
ident	morton	pth	0.418	0.002	13872	1	0	0
ident	morton	simd	0.6295	0.026	13832	1	0	0
ident	row	adv	0.0283	0.001	8204	1	0	0
ident	row	mpi	0.0219	0.001	8044	1	0	0

Continued on next page

Configuration-Level Factorial Raw Results (continued)

M	L	K	Mean s	SD s	RSS KiB	Sweeps	Rel. L2	Max corr.
ident	row	naive	0.0241	0.002	8048	1	0	0
ident	row	pth	0.0394	0.003	8108	1	0	0
ident	row	simd	0.0206	7.325e-04	8100	1	0	0
ident	stride-row	adv	0.0315	0.005	8024	1	0	0
ident	stride-row	mpi	0.0229	0.002	8088	1	0	0
ident	stride-row	naive	0.0203	4.718e-04	8048	1	0	0
ident	stride-row	pth	0.0388	0.004	7976	1	0	0
ident	stride-row	simd	0.0224	6.578e-04	8208	1	0	0
ill-cond	block-row	adv	1.5095	0.024	12096	22	8.252e-14	9.999e-13
ill-cond	block-row	mpi	0.9944	0.011	12252	27	3.497e-14	9.941e-13
ill-cond	block-row	naive	0.8184	0.029	12256	23	2.882e-14	9.991e-13
ill-cond	block-row	pth	0.785	0.016	12168	27	3.497e-14	9.941e-13
ill-cond	block-row	simd	0.8234	0.029	12252	23	2.882e-14	9.991e-13
ill-cond	col	adv	0.2632	0.004	12092	22	8.252e-14	9.999e-13
ill-cond	col	mpi	0.1782	0.004	12256	27	3.497e-14	9.941e-13
ill-cond	col	naive	0.1543	0.003	12096	23	2.882e-14	9.991e-13
ill-cond	col	pth	0.5691	0.003	12164	27	3.497e-14	9.941e-13
ill-cond	col	simd	0.1596	0.013	12092	23	2.882e-14	9.991e-13
ill-cond	jagged-row	adv	0.4326	0.005	12548	22	8.252e-14	9.999e-13
ill-cond	jagged-row	mpi	0.3117	0.004	12816	27	3.497e-14	9.941e-13
ill-cond	jagged-row	naive	0.2622	0.001	12752	23	2.882e-14	9.991e-13
ill-cond	jagged-row	pth	0.7694	0.015	12732	27	3.497e-14	9.941e-13
ill-cond	jagged-row	simd	0.2678	0.002	12540	23	2.882e-14	9.991e-13
ill-cond	morton	adv	16.6189	0.672	15236	22	8.252e-14	9.999e-13
ill-cond	morton	mpi	11.2033	0.21	15084	27	3.497e-14	9.941e-13
ill-cond	morton	naive	9.3452	0.279	15084	23	2.882e-14	9.991e-13
ill-cond	morton	pth	2.4985	0.068	15240	27	3.497e-14	9.941e-13
ill-cond	morton	simd	9.6633	0.126	15300	23	2.882e-14	9.991e-13
ill-cond	row	adv	0.5329	0.003	12096	22	8.252e-14	9.999e-13
ill-cond	row	mpi	0.4067	0.006	12256	27	3.497e-14	9.941e-13
ill-cond	row	naive	0.3566	0.012	12092	23	2.882e-14	9.991e-13
ill-cond	row	pth	0.718	0.006	12340	27	3.497e-14	9.941e-13
ill-cond	row	simd	0.3434	0.005	12352	23	2.882e-14	9.991e-13
ill-cond	stride-row	adv	0.5269	0.007	12144	22	8.252e-14	9.999e-13
ill-cond	stride-row	mpi	0.3954	0.009	12256	27	3.497e-14	9.941e-13
ill-cond	stride-row	naive	0.3483	0.012	11976	23	2.882e-14	9.991e-13
ill-cond	stride-row	pth	0.7193	0.015	12160	27	3.497e-14	9.941e-13
ill-cond	stride-row	simd	0.3553	0.023	12256	23	2.882e-14	9.991e-13
integer	block-row	adv	0.5945	0.013	11136	8	1.694e-14	9.985e-13
integer	block-row	mpi	0.4516	0.016	11136	10	2.025e-14	9.988e-13
integer	block-row	naive	0.4464	0.011	11136	10	2.363e-14	9.993e-13
integer	block-row	pth	0.3819	0.009	11220	10	2.025e-14	9.988e-13
integer	block-row	simd	0.4452	0.01	11132	10	2.363e-14	9.993e-13
integer	col	adv	0.1207	0.002	11132	8	1.694e-14	9.985e-13
integer	col	mpi	0.0931	0.002	11132	10	2.025e-14	9.988e-13
integer	col	naive	0.0922	0.002	11136	10	2.363e-14	9.993e-13
integer	col	pth	0.2398	0.003	11312	10	2.025e-14	9.988e-13
integer	col	simd	0.0944	0.002	11132	10	2.363e-14	9.993e-13
integer	jagged-row	adv	0.1759	0.001	11700	8	1.694e-14	9.985e-13
integer	jagged-row	mpi	0.1626	0.002	11700	10	2.025e-14	9.988e-13
integer	jagged-row	naive	0.1468	0.003	11420	10	2.363e-14	9.993e-13
integer	jagged-row	pth	0.374	0.03	11840	10	2.025e-14	9.988e-13
integer	jagged-row	simd	0.1481	7.707e-04	11424	10	2.363e-14	9.993e-13
integer	morton	adv	6.3265	0.037	13992	8	1.694e-14	9.985e-13
integer	morton	mpi	4.9021	0.076	13964	10	2.025e-14	9.988e-13
integer	morton	naive	4.7856	0.116	14204	10	2.363e-14	9.993e-13
integer	morton	pth	1.261	0.054	14152	10	2.025e-14	9.988e-13
integer	morton	simd	4.8403	0.078	14172	10	2.363e-14	9.993e-13
integer	row	adv	0.2258	0.005	11136	8	1.694e-14	9.985e-13
integer	row	mpi	0.2037	0.006	10860	10	2.025e-14	9.988e-13
integer	row	naive	0.1974	0.013	11132	10	2.363e-14	9.993e-13
integer	row	pth	0.3565	0.015	11044	10	2.025e-14	9.988e-13
integer	row	simd	0.1945	0.005	11136	10	2.363e-14	9.993e-13
integer	stride-row	adv	0.2303	0.006	11132	8	1.694e-14	9.985e-13
integer	stride-row	mpi	0.2059	0.003	11132	10	2.025e-14	9.988e-13
integer	stride-row	naive	0.1979	0.004	11136	10	2.363e-14	9.993e-13
integer	stride-row	pth	0.3514	0.003	11044	10	2.025e-14	9.988e-13
integer	stride-row	simd	0.1921	0.007	11136	10	2.363e-14	9.993e-13
lognorm	block-row	adv	0.569	0.021	12032	8	2.024e-14	9.933e-13
lognorm	block-row	mpi	0.4474	0.007	12252	10	2.867e-14	9.988e-13
lognorm	block-row	naive	0.4462	0.007	12144	10	2.730e-14	9.989e-13
lognorm	block-row	pth	0.3737	0.018	12148	10	2.867e-14	9.988e-13
lognorm	block-row	simd	0.4368	0.016	12352	10	2.730e-14	9.989e-13
lognorm	col	adv	0.114	9.522e-04	11984	8	2.024e-14	9.933e-13
lognorm	col	mpi	0.0918	0.004	12256	10	2.867e-14	9.988e-13
lognorm	col	naive	0.0881	0.003	12092	10	2.730e-14	9.989e-13
lognorm	col	pth	0.2257	0.005	12512	10	2.867e-14	9.988e-13
lognorm	col	simd	0.0864	0.001	12256	10	2.730e-14	9.989e-13
lognorm	jagged-row	adv	0.1805	0.002	12660	8	2.024e-14	9.933e-13
lognorm	jagged-row	mpi	0.1572	0.002	12820	10	2.867e-14	9.988e-13
lognorm	jagged-row	naive	0.1527	0.002	12656	10	2.730e-14	9.989e-13
lognorm	jagged-row	pth	0.3604	0.014	12720	10	2.867e-14	9.988e-13
lognorm	jagged-row	simd	0.1506	0.001	12656	10	2.730e-14	9.989e-13
lognorm	morton	adv	6.334	0.104	15324	8	2.024e-14	9.933e-13
lognorm	morton	mpi	4.9272	0.029	15080	10	2.867e-14	9.988e-13
lognorm	morton	naive	4.8927	0.097	15156	10	2.730e-14	9.989e-13
lognorm	morton	pth	1.2636	0.019	15412	10	2.867e-14	9.988e-13
lognorm	morton	simd	4.9427	0.032	15292	10	2.730e-14	9.989e-13

Continued on next page

Configuration-Level Factorial Raw Results (continued)

M	L	K	Mean s	SD s	RSS KiB	Sweeps	Rel. L2	Max corr.
lognorm	row	adv	0.2218	0.002	12252	8	2.024e-14	9.933e-13
lognorm	row	mpi	0.2049	0.005	12256	10	2.867e-14	9.988e-13
lognorm	row	naive	0.2005	0.003	12092	10	2.730e-14	9.989e-13
lognorm	row	pth	0.3329	0.01	12196	10	2.867e-14	9.988e-13
lognorm	row	simd	0.1977	0.004	12092	10	2.730e-14	9.989e-13
lognorm	stride-row	adv	0.2228	0.005	12096	8	2.024e-14	9.933e-13
lognorm	stride-row	mpi	0.2086	0.006	11980	10	2.867e-14	9.988e-13
lognorm	stride-row	naive	0.1966	0.006	12252	10	2.730e-14	9.989e-13
lognorm	stride-row	pth	0.3335	0.01	12144	10	2.867e-14	9.988e-13
lognorm	stride-row	simd	0.2002	0.004	11980	10	2.730e-14	9.989e-13
low-rank	block-row	adv	1.6081	0.008	12252	23	4.384e-14	9.964e-13
low-rank	block-row	mpi	0.7023	0.007	12256	17	3.100e-14	9.994e-13
low-rank	block-row	naive	0.6512	0.018	12256	16	2.422e-14	9.987e-13
low-rank	block-row	pth	0.587	0.018	12364	17	3.100e-14	9.994e-13
low-rank	block-row	simd	0.6345	0.017	11980	16	2.422e-14	9.987e-13
low-rank	col	adv	0.2817	0.004	12256	23	4.384e-14	9.964e-13
low-rank	col	mpi	0.1384	0.008	12340	17	3.100e-14	9.994e-13
low-rank	col	naive	0.1205	0.002	12256	16	2.422e-14	9.987e-13
low-rank	col	pth	0.3692	0.009	12148	17	3.100e-14	9.994e-13
low-rank	col	simd	0.1194	0.001	12252	16	2.422e-14	9.987e-13
low-rank	jagged-row	adv	0.4963	0.005	12900	23	4.384e-14	9.964e-13
low-rank	jagged-row	mpi	0.2456	0.003	12816	17	3.100e-14	9.994e-13
low-rank	jagged-row	naive	0.2217	0.003	12544	16	2.422e-14	9.987e-13
low-rank	jagged-row	pth	0.5782	0.003	12716	17	3.100e-14	9.994e-13
low-rank	jagged-row	simd	0.2233	0.006	12820	16	2.422e-14	9.987e-13
low-rank	morton	adv	18.3116	0.095	15324	23	4.384e-14	9.964e-13
low-rank	morton	mpi	8.0842	0.057	15300	17	3.100e-14	9.994e-13
low-rank	morton	naive	7.4948	0.048	15080	16	2.422e-14	9.987e-13
low-rank	morton	pth	1.882	0.035	15280	17	3.100e-14	9.994e-13
low-rank	morton	simd	7.6761	0.127	15292	16	2.422e-14	9.987e-13
low-rank	row	adv	0.6208	0.01	12252	23	4.384e-14	9.964e-13
low-rank	row	mpi	0.3321	0.015	12256	17	3.100e-14	9.994e-13
low-rank	row	naive	0.2899	0.004	12256	16	2.422e-14	9.987e-13
low-rank	row	pth	0.5365	0.005	12380	17	3.100e-14	9.994e-13
low-rank	row	simd	0.3176	0.019	12256	16	2.422e-14	9.987e-13
low-rank	stride-row	adv	0.6295	0.007	12256	23	4.384e-14	9.964e-13
low-rank	stride-row	mpi	0.3312	0.01	11976	17	3.100e-14	9.994e-13
low-rank	stride-row	naive	0.3033	0.01	12336	16	2.422e-14	9.987e-13
low-rank	stride-row	pth	0.5445	0.015	12152	17	3.100e-14	9.994e-13
low-rank	stride-row	simd	0.2988	0.012	11980	16	2.422e-14	9.987e-13
normal	block-row	adv	0.5676	0.019	12252	8	1.770e-14	9.932e-13
normal	block-row	mpi	0.4437	0.026	12252	10	2.158e-14	9.999e-13
normal	block-row	naive	0.4458	0.005	12256	10	2.478e-14	9.975e-13
normal	block-row	pth	0.3708	0.006	12108	10	2.158e-14	9.999e-13
normal	block-row	simd	0.4353	0.005	12256	10	2.478e-14	9.975e-13
normal	col	adv	0.1097	0.002	12256	8	1.770e-14	9.932e-13
normal	col	mpi	0.0842	0.004	11980	10	2.158e-14	9.999e-13
normal	col	naive	0.0836	0.002	12252	10	2.478e-14	9.975e-13
normal	col	pth	0.2288	0.004	12420	10	2.158e-14	9.999e-13
normal	col	simd	0.0873	0.001	12256	10	2.478e-14	9.975e-13
normal	jagged-row	adv	0.1817	0.003	12820	8	1.770e-14	9.932e-13
normal	jagged-row	mpi	0.1551	0.002	12816	10	2.158e-14	9.999e-13
normal	jagged-row	naive	0.1527	0.001	12820	10	2.478e-14	9.975e-13
normal	jagged-row	pth	0.3575	0.01	12888	10	2.158e-14	9.999e-13
normal	jagged-row	simd	0.1534	0.002	12816	10	2.478e-14	9.975e-13
normal	morton	adv	6.4074	0.193	15180	8	1.770e-14	9.932e-13
normal	morton	mpi	4.8999	0.022	15296	10	2.158e-14	9.999e-13
normal	morton	naive	4.7788	0.134	15296	10	2.478e-14	9.975e-13
normal	morton	pth	1.2238	0.039	15460	10	2.158e-14	9.999e-13
normal	morton	simd	4.9361	0.063	15296	10	2.478e-14	9.975e-13
normal	row	adv	0.2302	0.004	12256	8	1.770e-14	9.932e-13
normal	row	mpi	0.2034	0.013	12256	10	2.158e-14	9.999e-13
normal	row	naive	0.2025	0.002	12256	10	2.478e-14	9.975e-13
normal	row	pth	0.3095	0.008	12192	10	2.158e-14	9.999e-13
normal	row	simd	0.186	0.002	12252	10	2.478e-14	9.975e-13
normal	stride-row	adv	0.2246	0.002	12252	8	1.770e-14	9.932e-13
normal	stride-row	mpi	0.2016	0.01	12256	10	2.158e-14	9.999e-13
normal	stride-row	naive	0.1998	0.005	12252	10	2.478e-14	9.975e-13
normal	stride-row	pth	0.3411	0.006	12372	10	2.158e-14	9.999e-13
normal	stride-row	simd	0.2024	0.003	12256	10	2.478e-14	9.975e-13
radem	block-row	adv	0.6046	0.02	10900	8	1.778e-14	9.996e-13
radem	block-row	mpi	0.4098	0.01	11132	9	2.165e-14	9.982e-13
radem	block-row	naive	0.4344	0.007	11136	10	2.429e-14	9.999e-13
radem	block-row	pth	0.3443	0.003	11032	9	2.165e-14	9.982e-13
radem	block-row	simd	0.4391	0.014	11132	10	2.429e-14	9.999e-13
radem	col	adv	0.1158	0.002	11132	8	1.778e-14	9.996e-13
radem	col	mpi	0.0816	0.001	10860	9	2.165e-14	9.982e-13
radem	col	naive	0.0869	0.002	11132	10	2.429e-14	9.999e-13
radem	col	pth	0.2046	0.005	11020	9	2.165e-14	9.982e-13
radem	col	simd	0.0874	0.002	11136	10	2.429e-14	9.999e-13
radem	jagged-row	adv	0.1743	0.006	11696	8	1.778e-14	9.996e-13
radem	jagged-row	mpi	0.1344	0.001	11696	9	2.165e-14	9.982e-13
radem	jagged-row	naive	0.1452	0.002	11700	10	2.429e-14	9.999e-13
radem	jagged-row	pth	0.3136	0.006	11604	9	2.165e-14	9.982e-13
radem	jagged-row	simd	0.139	0.003	11700	10	2.429e-14	9.999e-13
radem	morton	adv	6.2599	0.082	13988	8	1.778e-14	9.996e-13
radem	morton	mpi	4.4668	0.016	14176	9	2.165e-14	9.982e-13
radem	morton	naive	4.7695	0.031	14180	10	2.429e-14	9.999e-13

Continued on next page

Configuration-Level Factorial Raw Results (continued)

M	L	K	Mean s	SD s	RSS KiB	Sweeps	Rel. L2	Max corr.
radem	morton	pth	1.1767	0.018	14184	9	2.165e-14	9.982e-13
radem	morton	simd	4.8572	0.058	13964	10	2.429e-14	9.999e-13
radem	row	adv	0.2182	0.005	11132	8	1.778e-14	9.996e-13
radem	row	mpi	0.1841	0.004	11132	9	2.165e-14	9.982e-13
radem	row	naive	0.195	0.005	11132	10	2.429e-14	9.999e-13
radem	row	pth	0.3097	0.014	11236	9	2.165e-14	9.982e-13
radem	row	simd	0.1911	0.008	11136	10	2.429e-14	9.999e-13
radem	stride-row	adv	0.2212	0.005	10864	8	1.778e-14	9.996e-13
radem	stride-row	mpi	0.1936	0.01	11136	9	2.165e-14	9.982e-13
radem	stride-row	naive	0.193	0.005	11132	10	2.429e-14	9.999e-13
radem	stride-row	pth	0.3034	0.006	11036	9	2.165e-14	9.982e-13
radem	stride-row	simd	0.1927	0.005	11132	10	2.429e-14	9.999e-13
sparse	block-row	adv	0.5385	0.008	11132	7	1.729e-14	9.890e-13
sparse	block-row	mpi	0.4158	0.008	11132	9	2.094e-14	9.983e-13
sparse	block-row	naive	0.4109	0.003	10972	9	2.146e-14	1.000e-12
sparse	block-row	pth	0.359	0.009	11048	9	2.094e-14	9.983e-13
sparse	block-row	simd	0.4063	0.002	11136	9	2.146e-14	1.000e-12
sparse	col	adv	0.1091	0.003	11132	7	1.729e-14	9.890e-13
sparse	col	mpi	0.087	0.002	11132	9	2.094e-14	9.983e-13
sparse	col	naive	0.0874	0.003	11136	9	2.146e-14	1.000e-12
sparse	col	pth	0.2019	0.005	11296	9	2.094e-14	9.983e-13
sparse	col	simd	0.0813	0.001	10972	9	2.146e-14	1.000e-12
sparse	jagged-row	adv	0.1648	0.003	11536	7	1.729e-14	9.890e-13
sparse	jagged-row	mpi	0.1435	0.002	11696	9	2.094e-14	9.983e-13
sparse	jagged-row	naive	0.1424	0.003	11700	9	2.146e-14	1.000e-12
sparse	jagged-row	pth	0.3264	0.006	11800	9	2.094e-14	9.983e-13
sparse	jagged-row	simd	0.1442	0.004	11700	9	2.146e-14	1.000e-12
sparse	morton	adv	5.627	0.045	14200	7	1.729e-14	9.890e-13
sparse	morton	mpi	4.4682	0.031	14176	9	2.094e-14	9.983e-13
sparse	morton	naive	4.4522	0.034	14176	9	2.146e-14	1.000e-12
sparse	morton	pth	1.1656	0.018	13988	9	2.094e-14	9.983e-13
sparse	morton	simd	4.5516	0.031	13964	9	2.146e-14	1.000e-12
sparse	row	adv	0.2028	0.007	10860	7	1.729e-14	9.890e-13
sparse	row	mpi	0.1833	0.002	11136	9	2.094e-14	9.983e-13
sparse	row	naive	0.2005	0.009	11136	9	2.146e-14	1.000e-12
sparse	row	pth	0.2978	0.004	11240	9	2.094e-14	9.983e-13
sparse	row	simd	0.187	0.003	11132	9	2.146e-14	1.000e-12
sparse	stride-row	adv	0.2027	0.002	11136	7	1.729e-14	9.890e-13
sparse	stride-row	mpi	0.1863	0.003	11132	9	2.094e-14	9.983e-13
sparse	stride-row	naive	0.1779	0.004	11132	9	2.146e-14	1.000e-12
sparse	stride-row	pth	0.305	0.006	11180	9	2.094e-14	9.983e-13
sparse	stride-row	simd	0.1891	0.004	11136	9	2.146e-14	1.000e-12
uniform	block-row	adv	0.5997	0.006	12176	8	1.786e-14	9.976e-13
uniform	block-row	mpi	0.4521	0.005	12256	10	2.019e-14	9.993e-13
uniform	block-row	naive	0.4539	0.027	12256	10	2.336e-14	9.946e-13
uniform	block-row	pth	0.3796	0.004	12344	10	2.019e-14	9.993e-13
uniform	block-row	simd	0.4469	0.005	12256	10	2.336e-14	9.946e-13
uniform	col	adv	0.1183	9.078e-04	12172	8	1.786e-14	9.976e-13
uniform	col	mpi	0.0922	0.003	12252	10	2.019e-14	9.993e-13
uniform	col	naive	0.0909	0.001	12276	10	2.336e-14	9.946e-13
uniform	col	pth	0.2506	0.005	12372	10	2.019e-14	9.993e-13
uniform	col	simd	0.0974	0.004	12252	10	2.336e-14	9.946e-13
uniform	jagged-row	adv	0.1887	0.004	12816	8	1.786e-14	9.976e-13
uniform	jagged-row	mpi	0.1556	0.003	12820	10	2.019e-14	9.993e-13
uniform	jagged-row	naive	0.143	0.004	12704	10	2.336e-14	9.946e-13
uniform	jagged-row	pth	0.3434	0.004	12904	10	2.019e-14	9.993e-13
uniform	jagged-row	simd	0.1468	9.092e-04	12820	10	2.336e-14	9.946e-13
uniform	morton	adv	6.6109	0.078	15324	8	1.786e-14	9.976e-13
uniform	morton	mpi	4.8138	0.141	15464	10	2.019e-14	9.993e-13
uniform	morton	naive	4.8088	0.07	15296	10	2.336e-14	9.946e-13
uniform	morton	pth	1.2854	0.024	15404	10	2.019e-14	9.993e-13
uniform	morton	simd	4.8662	0.137	15240	10	2.336e-14	9.946e-13
uniform	row	adv	0.2329	0.005	12256	8	1.786e-14	9.976e-13
uniform	row	mpi	0.2106	0.007	12252	10	2.019e-14	9.993e-13
uniform	row	naive	0.2107	0.017	12140	10	2.336e-14	9.946e-13
uniform	row	pth	0.3322	0.01	12500	10	2.019e-14	9.993e-13
uniform	row	simd	0.2008	0.005	12256	10	2.336e-14	9.946e-13
uniform	stride-row	adv	0.2356	0.003	12256	8	1.786e-14	9.976e-13
uniform	stride-row	mpi	0.2095	0.003	12252	10	2.019e-14	9.993e-13
uniform	stride-row	naive	0.2018	0.002	12256	10	2.336e-14	9.946e-13
uniform	stride-row	pth	0.3045	0.008	12312	10	2.019e-14	9.993e-13
uniform	stride-row	simd	0.1867	0.003	12252	10	2.336e-14	9.946e-13
zero	block-row	adv	0.1	0.002	8204	1	0	0
zero	block-row	mpi	0.0732	0.001	8040	1	0	0
zero	block-row	naive	0.078	0.002	7808	1	0	0
zero	block-row	pth	0.0722	0.002	7988	1	0	0
zero	block-row	simd	0.0744	1.301e-04	8204	1	0	0
zero	col	adv	0.0235	0.001	8044	1	0	0
zero	col	mpi	0.0187	3.929e-04	8032	1	0	0
zero	col	naive	0.0192	6.743e-04	8048	1	0	0
zero	col	pth	0.0366	0.001	8340	1	0	0
zero	col	simd	0.0213	9.297e-04	8032	1	0	0
zero	jagged-row	adv	0.022	0.002	8580	1	0	0
zero	jagged-row	mpi	0.0196	0.002	8588	1	0	0
zero	jagged-row	naive	0.0171	6.408e-04	8576	1	0	0
zero	jagged-row	pth	0.0319	6.128e-04	8732	1	0	0
zero	jagged-row	simd	0.017	4.154e-04	8756	1	0	0
zero	morton	adv	0.8315	0.004	14044	1	0	0

Continued on next page

Configuration-Level Factorial Raw Results (continued)

M	L	K	Mean s	SD s	RSS KiB	Sweeps	Rel. L2	Max corr.
zero	morton	mpi	0.5385	0.009	14016	1	0	0
zero	morton	naive	0.5323	0.011	14012	1	0	0
zero	morton	pth	0.3237	0.004	14000	1	0	0
zero	morton	simd	0.5202	0.021	14016	1	0	0
zero	row	adv	0.0256	0.002	8196	1	0	0
zero	row	mpi	0.0179	0.001	8032	1	0	0
zero	row	naive	0.0197	0.001	8088	1	0	0
zero	row	pth	0.0319	9.455e-04	8020	1	0	0
zero	row	simd	0.0185	0.001	8048	1	0	0
zero	stride-row	adv	0.0276	0.001	8048	1	0	0
zero	stride-row	mpi	0.0177	5.626e-04	8044	1	0	0
zero	stride-row	naive	0.0229	0.002	8036	1	0	0
zero	stride-row	pth	0.0316	0.001	8036	1	0	0
zero	stride-row	simd	0.0186	0.001	8208	1	0	0

TABLE XIII: Focused Profiler Raw Results: Instruction and Branch Tools

Tool	M	L	K	Dur. s	Ir	Bc	Bcm	Sweeps
cachegrind	cauchy	col	simd	2.076	1027753535	105870603	1063132	10
cachegrind	hilbert	row	adv	10.143	6012238469	719292430	3041758	23
cachegrind	ill-cond	row	adv	9.086	5399839223	666771193	3030465	22
cachegrind	low-rank	row	adv	10.246	6041301507	720575334	3171285	23
cachegrind	normal	col	naive	2.221	1070778212	108847386	985326	10
cachegrind	normal	col	simd	2.069	1071023531	108847584	985334	10
cachegrind	normal	morton	adv	144.917	110197853348	5434003879	569246977	8
cachegrind	normal	morton	naive	112.361	84906406769	4131970885	428971481	10
cachegrind	normal	row	adv	4.075	2143231776	263350850	1490216	8
cachegrind	normal	row	naive	3.322	1699531579	192469148	985246	10
cachegrind	zero	morton	adv	18.169	14571599840	702932245	67610438	1
cachegrind	zero	row	naive	0.816	184842754	27832323	147590	1
callgrind	cauchy	col	simd	2.223	1026900421	–	–	10
callgrind	hilbert	row	adv	12.998	6010831251	–	–	23
callgrind	ill-cond	row	adv	11.953	5398687415	–	–	22
callgrind	low-rank	row	adv	12.947	6039968863	–	–	23
callgrind	normal	col	naive	2.27	1069889263	–	–	10
callgrind	normal	col	simd	2.276	1070133234	–	–	10
callgrind	normal	morton	adv	84.868	110197200419	–	–	8
callgrind	normal	morton	naive	64.529	84905753907	–	–	10
callgrind	normal	row	adv	5.279	2142377711	–	–	8
callgrind	normal	row	naive	4.138	1698642648	–	–	10
callgrind	zero	morton	adv	11.242	14571259419	–	–	1
callgrind	zero	row	naive	0.916	184502433	–	–	1

TABLE XIV: Focused Profiler Raw Results: Heap Tools

Tool	M	L	K	Dur. s	Massif peak B	DHAT total B	DHAT live B
heaptrack	normal	block-row	naive	0.639	–	–	–
heaptrack	normal	col	adv	0.265	–	–	–
heaptrack	normal	col	naive	0.266	–	–	–
heaptrack	normal	col	pth	0.366	–	–	–
heaptrack	normal	jagged-row	naive	0.314	–	–	–
heaptrack	normal	morton	adv	6.538	–	–	–
heaptrack	normal	morton	naive	4.925	–	–	–
heaptrack	normal	morton	pth	1.418	–	–	–
heaptrack	normal	row	adv	0.365	–	–	–
heaptrack	normal	row	naive	0.366	–	–	–
heaptrack	normal	row	pth	0.465	–	–	–
heaptrack	normal	stride-row	naive	0.365	–	–	–
heaptrack	zero	col	adv	0.164	–	–	–
heaptrack	zero	morton	adv	0.967	–	–	–
heaptrack	zero	row	adv	0.265	–	–	–
dhat	normal	block-row	naive	2.597	–	9441187	2949122
dhat	normal	col	adv	1.769	–	12040995	1966081
dhat	normal	col	naive	1.468	–	9440011	2949122
dhat	normal	col	pth	2.269	–	11052704	2949122
dhat	normal	jagged-row	naive	11.5	–	9467699	2949122
dhat	normal	morton	adv	15.654	–	21608235	2097152
dhat	normal	morton	naive	11.891	–	19007251	2097152
dhat	normal	morton	pth	13.161	–	20619944	2097152
dhat	normal	row	adv	2.72	–	12040971	1966081
dhat	normal	row	naive	2.321	–	9439987	2949122
dhat	normal	row	pth	3.174	–	11052680	2949122
dhat	normal	stride-row	naive	2.272	–	9440107	2949122
dhat	zero	col	adv	0.816	–	6323192	524800
dhat	zero	morton	adv	2.63	–	15890432	2097152
dhat	zero	row	adv	0.866	–	6323168	524800
massif	normal	block-row	naive	2.114	3701568	–	–
massif	normal	col	adv	0.816	3700352	–	–
massif	normal	col	naive	0.77	3698688	–	–
massif	normal	col	pth	1.467	3719776	–	–

Continued on next page

Focused Profiler Raw Results: Heap Tools (continued)

Tool	M	L	K	Dur. s	Massif peak B	DHAT total B	DHAT live B
massif	normal	jagged-row	naive	1.519	3708792	–	–
massif	normal	morton	adv	12.662	9010648	–	–
massif	normal	morton	naive	9.034	9010648	–	–
massif	normal	morton	pth	10.436	9029872	–	–
massif	normal	row	adv	2.12	3700288	–	–
massif	normal	row	naive	1.668	3700288	–	–
massif	normal	row	pth	2.57	3719152	–	–
massif	normal	stride-row	naive	1.671	3700432	–	–
massif	zero	col	adv	0.565	2719176	–	–
massif	zero	morton	adv	2.022	9010648	–	–
massif	zero	row	adv	0.666	2719080	–	–

TABLE XV: Focused Profiler Raw Results: mpiP Runs

M	L	K	NP	Dur. s	App s	MPI %	MPI s
hilbert	row	mpi	2	0.866	0.962	49.81	0.479
hilbert	row	mpi	4	0.869	2.04	74.59	1.52
low-rank	col	mpi	2	0.516	0.3	49.29	0.148
low-rank	row	mpi	2	0.716	0.683	49.6	0.339
normal	col	mpi	2	0.465	0.192	48.91	0.094
normal	col	mpi	4	0.468	0.385	72.25	0.278
normal	morton	mpi	2	5.631	10.5	49.97	5.24
normal	morton	mpi	4	5.981	22.5	74.95	16.8
normal	row	mpi	2	0.566	0.45	49.56	0.223
normal	row	mpi	4	0.621	0.886	73.76	0.653
zero	row	mpi	2	0.365	0.035	47.53	0.017
zero	row	mpi	4	0.416	0.077	70.91	0.055