An Efficient 2D Fusion Method for High-Performance Two-Stage Eigensolvers on Modern Heterogeneous Architectures

Yongxiao Zhou
Tsinghua University
Beijing, China
zhou-yx23@mails.tsinghua.edu.cn

Yi Zong
Tsinghua University
Beijing, China
zong-y21@mails.tsinghua.edu.cn

Yuyang Jin
Tsinghua University
Beijing, China
jinyuyang@tsinghua.edu.cn

Heng Li
Tsinghua University
Beijing, China
liheng19@mails.tsinghua.edu.cn

Wei Xue
Tsinghua University, Beijing, China; Qinghai University, Xining, China
Beijing, China
xuewei@tsinghua.edu.cn

Abstract

Solving a significant portion of the eigensystem is a critical problem in numerical linear algebra and is widely applied in real-world applications. As problem sizes increase, the two-stage tridiagonalization method has emerged as the state-of-the-art approach and has been implemented in well-known libraries such as LAPACK, PLASMA, and MAGMA. Its major performance bottleneck is the tridiagonal-to-band back transformation of eigenvectors (*st2sb*) due to the dilemma between limited operational intensity and excessive computational cost. This challenge is further exacerbated by the growing imbalance between computational speed and memory bandwidth in modern heterogeneous architectures.

To address this challenge, this paper introduces a 2D Fusion method to decouple the operational intensity from the computational cost of *st2sb*. To reduce the intrinsic overhead of 2D Fusion for large fusion factors, we further propose an effective skipping strategy. Our 2D Fusion enhances the performance of all existing two-stage eigensolvers without loss of accuracy. We evaluated the effectiveness of 2D Fusion in MAGMA and LAPACK across various problem sizes: on the Nvidia A100 GPU, 2D Fusion improves the performance of eigenvalue decomposition in MAGMA by an average speedup of **1.06**× for matrices larger than 24k×24k in FP64; on the SW26010-Pro, 2D Fusion accelerates the two-stage eigensolver tuned on LAPACK with an average speedup of **1.19**× for eigenvalue decomposition in both FP32 and FP64.

CCS Concepts

• **Mathematics of computing** → **Mathematical software performance**; **Solvers**; • **Computing methodologies** → **Parallel algorithms**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

Keywords

Eigenvectors, blocked Householder transformations, heterogeneous architectures, symmetric matrices

ACM Reference Format:
Yongxiao Zhou, Yi Zong, Yuyang Jin, Heng Li, and Wei Xue. 2025. An Efficient 2D Fusion Method for High-Performance Two-Stage Eigensolvers on Modern Heterogeneous Architectures. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3730411

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730411

1 Introduction

Eigenvalue decomposition (EVD) is a fundamental problem in linear algebra and plays a crucial role in various fields, including scientific computing [32], data analysis [16], and engineering [26]. Among standard eigenproblems, solving a large fraction of eigensystems for dense symmetric matrices is one of the most typical scenarios [4, 28], which is also a core functionality of LAPACK [2]. However, the eigensystem calculation of large-scale matrices is often a

performance bottleneck in applications due to its high complexity of $O(N^3)$ [20]. For instance, it can account for over 90% of the total time in the LS3DF algorithm for 8,000 silicon atoms in electronic structure calculations [46]. Therefore, developing a high-performance eigensolver is essential to meet the growing computational demands.

Methods for solving symmetric eigenproblems can be categorized into four types: subspace iteration methods [38], Jacobi methods [6], the one-stage tridiagonalization method [17], and the two-stage tridiagonalization method [9]. Subspace iteration methods are inefficient when computing a large number of eigenvectors, as they primarily rely on BLAS-2 matrix-vector multiplications rather than BLAS-3 operations [12]. Meanwhile, Jacobi methods are impractical for large matrices due to their prohibitively high computational cost [16]. The two-stage tridiagonalization method [9] has been proven more effective for large matrices by alleviating the memory bottleneck of the one-stage tridiagonalization method [2]. Currently, this two-stage method is implemented in well-established libraries such as SBR [11] and PLASMA [45] for CPU platforms, and MAGMA [42] and ELPA [32, 49] for heterogeneous architectures.

The tridiagonal-to-band back transformation for eigenvectors (st2sb) *has historically been the performance bottleneck in solving eigensystems of large-scale matrices with the two-stage tridiagonalization method due to its limited operational intensity (OI [47]) [20]. st2sb* can account for over 40% of the total runtime in eigenvalue decomposition of a matrix of size 130,000 on the K computer [20]. Existing two-stage eigensolvers [11, 32, 42, 45, 49] improve the OI of *st2sb* through **1D Fusion**. They fuse multiple Householder reflectors into a blocked Householder transformation [18] and update the eigenvectors via BLAS-3 matrix-matrix multiplications (GEMM) instead of BLAS-2 kernels.

However, 1D fusion enhances the OI of *st2sb* at the cost of a linear increase in computational workload. As analyzed in Section 2.1, the best time lower bound of *st2sb* is achieved when its OI matches the machine balance (defined as the ratio of peak computational speed to peak memory bandwidth [47]), and the GEMM performance reaches the theoretical peak. As illustrated in Figure 1, the corresponding computational cost can substantially exceed $4N^3$ for low-precision computations on modern heterogeneous architectures, due to the increase in machine balance from $O(10)$ to $O(100)$ driven by the emergence of powerful computing units such as Tensor Cores [36] on Nvidia GPUs and Matrix Cores [1] on AMD GPUs. This high computational cost makes *st2sb* the primary performance bottleneck in two-stage EVD. As the gap between computational throughput and memory bandwidth continues to widen by approximately 50% per year [39], a similar situation is expected for FP32 and FP64 computations, as suggested by the extrapolated trend in the lower-left region of Figure 1. Moreover, even when the OI matches the machine balance, the actual GEMM performance still exhibits a substantial gap from the theoretical peak for all precisions due to overheads in BLAS implementations [12], such as packing costs, pipeline stalls, memory access inefficiencies, and the cost of parallelization. However, achieving both high OI and low computational cost in the *st2sb* step of two-stage eigensolvers remains an open challenge in the community, which is the most critical issue for improving EVD performance [16, 20].

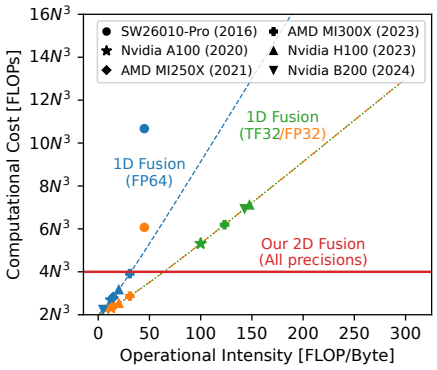

Figure 1: Computational cost comparison of the tridiagonal-to-band back transformation to achieve the same operational intensity as the machine balance: 1D Fusion (scatter points) vs. our 2D Fusion (the red horizontal line). The matrix bandwidth is 256 for all architectures except SW26010-Pro [21], where it is 96 due to the limited size of Local Data Memory. Further details are provided in Section 2.1.

To address this challenge, we propose the following contributions:

- We propose a **2D Fusion** method that decouples the OI from the computational cost of *st2sb* by further introducing the new fusion of blocked Householder transformations in 1D Fusion. We demonstrate that the OI of *st2sb* with 2D Fusion is unbounded, and by tuning the fusion factor, the computational cost can be limited to approximately $4N^3$. It highlights the superiority of 2D Fusion over 1D Fusion in developing high-performance eigensolvers on modern heterogeneous architectures.
- We propose a **skipping strategy** to reduce the intrinsic cost of 2D Fusion for large fusion factors by bypassing unnecessary matrix computations. Experimental results show that this strategy reduces the overhead of 2D Fusion by an average of 49.0% in FP32 and 59.8% in FP64 across various matrix sizes.

- We integrate our 2D Fusion method into both MAGMA and LAPACK. Tests with various problem sizes show that, on the Nvidia A100 GPU, 2D Fusion accelerates *st2sb* in MAGMA by an average speedup of **1.21**× for matrices larger than 24k×24k in FP64, resulting in an average EVD speedup of **1.06**×. Additionally, we employ a simple TF32+FP32 mixed-precision scheme to further evaluate the effectiveness of our 2D Fusion. On the SW26010-Pro [21], 2D Fusion accelerates *st2sb* of the two-stage eigensolver tuned on LAPACK with an average speedup of **1.94**×, leading to a **1.19**× EVD speedup in both FP32 and FP64.

2 The Two-stage Tridiagonalization Method

Bischof et al. [9] proposed the two-stage tridiagonalization method to address the memory bottleneck of the one-stage tridiagonalization method [17] by splitting the tridiagonalization into two stages. However, this approach incurs an additional overhead of the tridiagonal-to-band back transformation for eigenvectors, which often becomes the performance bottleneck due to its limited OI.

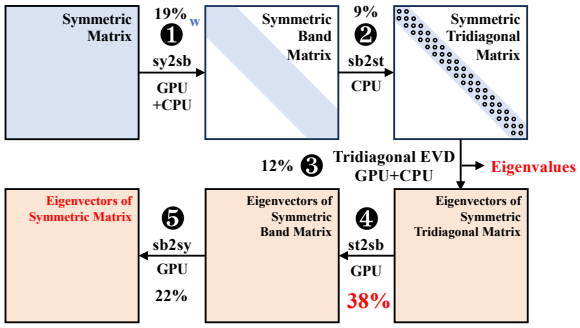

Figure 2: The workflow of current two-stage eigensolvers on CPU-GPU heterogeneous platforms. The percentages indicate the time breakdown in the EVD solving process for a matrix of size 40,960 on the Nvidia A100 GPU, with *st2sb* being the performance bottleneck.

As illustrated in Figure 2, the solving process of the two-stage EVD method consists of the following five steps:

❶ ***Symmetric-to-Band Reduction (sy2sb).*** In each step of *sy2sb*, QR factorization is first applied to a blocked panel of the original matrix. Then, the trailing matrix is efficiently updated using symmetric matrix multiplication (SYMM) and symmetric rank-2k updates (SYR2K), with a total computational cost of $\frac{4}{3}N^3$.

❷ ***Band-to-Tridiagonal Reduction (sb2st).*** The obtained band matrix is further reduced to a tridiagonal form through $N-2$ sweeps of Householder transformations, known as "bulge chasing". To achieve high performance, this memory-bound step requires a sufficiently small matrix bandwidth to fit the band matrix blocks into high-speed cache (e.g., the CPU's L2 data cache) [24].

❸ ***Solution of the tridiagonal eigenproblem (solution).*** This step typically accounts for only a small fraction of the total solution time. The commonly used methods include the Divide-and-Conquer method [23], the Bisection and Inverse Iteration method [33], and the Multiple Relatively Robust Representations (MR3) method [15].

❹ ***Tridiagonal-to-Band Back Transformation (st2sb).*** The $N^2/2w$ Householder transformations generated during *sb2st* must be applied to the eigenvectors of the tridiagonal matrix to recover the eigenvectors of the band matrix, where w is the matrix bandwidth. Each transformation affects w rows of eigenvectors with corresponding offsets, and the transformations must follow a specific order of dependencies that aligns with *sb2st*. Directly applying these Householder transformations for eigenvector recovery using BLAS-2 operations requires $O(N^3)$ memory accesses, which is obviously inefficient and becomes a performance bottleneck [20].

❺ ***Band-to-Symmetric Back Transformation (sb2sy).*** The sequence of orthogonal transformations from the *sy2sb* step are applied in reverse order to recover the eigenvectors of the original matrix. This is achieved using high-performance GEMM operations at a computational cost of $2N^3$.

Table 1 summarizes the kernel types and computational costs of these solution steps.

2.1 Limitations of 1D Fusion for Tridiagonal-to-Band Back Transformation

Currently, the two-stage tridiagonalization method has been implemented in SBR [11], PLASMA [45], MAGMA [26], and ELPA [32, 49], as summarized in Table 2. All these implementations employ a **1D Fusion** strategy to enhance the OI of *st2sb*, allowing the tridiagonal-to-band back transformation to be performed using BLAS-3 GEMM operations instead of the inefficient BLAS-2 operations. Specifically, multiple Householder reflectors are first zero-padded along the sweep direction and then fused into a blocked Householder transformation Q. The compact WY representation [41] of Q, given by $Q = I + WTW^T$, is then utilized to update the corresponding rows of eigenvectors via BLAS-3 GEMM operations.

*The primary challenge of the current 1D Fusion strategy lies in **balancing OI and overall computational cost** to accelerate st2sb.* Since the size of matrix bandwidth is constrained by the performance of *sb2st*, the only way to enhance the OI of *st2sb* is to increase the block size of 1D Fusion. However, a larger block size also leads to a higher computational cost due to increased overlapping row updates of the eigenvectors.

Table 1: Kernel types and computational costs of the two-stage tridiagonalization method (without 1D Fusion). *sy2sb* and *sb2st* denote the symmetric-to-band and band-to-tridiagonal reductions, respectively, while *st2sb* and *sb2sy* are the corresponding back-transformations for eigenvectors. w is the matrix bandwidth, and b is the block size. For BLAS-2 kernels, only memory operations are considered, whereas for BLAS-3 kernels, floating-point operations are measured.

Steps	Description	BLAS-2 kernels	Total Mem. ops.	BLAS-3 kernels	Total FP ops.
sy2sb	Panel QR factorization	GEMV, GER	$\frac{3}{4}N^2w$	/	/
	Compute compact WY representation [41]	GEMV	$\frac{1}{2}N^2w$	/	/
	Update trailing matrix	/	/	SYMM, SYR2K	$\frac{4}{3}N^3$
sb2st	One-sided updates in multiple sweeps	GEMV, GER	$3N^2w$	/	/
	Two-sided updates in multiple sweeps	SYMV, SYR2	$\frac{3}{4}N^2w$	/	/
solution	Tridiagonal EVD with eigenvectors (Divide-and-Conquer [23]) Q	MISC	/	GEMM	$\frac{4}{3}N^3$
st2sb	Update Q using the Householder reflectors generated during the *sb2st*	GEMV, GER	$\frac{3}{2}N^3$	/	/
sb2sy	Update Q with blocked Householder transformations from *sy2sb*	TRMV, GEMV	/	TRMM, GEMM	$2N^3$

Table 2: Summary of existing two-stage eigensolvers.

Libraries	Eigenvalues	Eigenvectors	***st2sb*** Fusion	Platform
LAPACK [2]	✓	×	/	CPU
SBR [11]	✓	✓	1D	CPU
PLASMA [25]	✓	✓	1D	CPU
ELPA [32, 49]	✓	✓	1D	CPU+GPU
MAGMA [26]	✓	✓	1D	CPU+GPU
Ours	✓	✓	**2D**	CPU+GPU, SW

This trade-off makes it challenging for 1D Fusion to further accelerate *st2sb*, particularly on modern heterogeneous architectures with high machine balance, as demonstrated in the following analysis.

Figure 3 illustrates how the eigenvectors are updated during *st2sb* with 1D Fusion with the matrix size N, the matrix bandwidth w, and the block size b. Vectors of length w in each column correspond to the Householder transformations applied during the same sweep of *sb2st*. These Householder reflectors are fused into **1D blocks** along the sweep direction with the block size b, as shown by the six black-bordered blocks of Householder reflectors in Figure 3. Each 1D block is then zero-padded into a matrix $W \in \mathbb{R}^{(w+b-1)\times b}$, after which its compact WY representation, $Q = I + WTW^T$, can be generated. Since computing the compact WY representations has relatively low computational cost, it is omitted in the following performance analysis. Subsequently, the corresponding rows of eigenvectors, $E_{\text{rows}} \in \mathbb{R}^{(w+b-1)\times N}$, are updated as $QE_{\text{rows}} = (I + WTW^T)E_{\text{rows}}$ in three GEMM operations:

$$X_1 = W^T E_{\text{rows}}, \tag{1}$$

$$X_2 = WT, \tag{2}$$

$$E_{\text{rows}} = E_{\text{rows}} + X_2 X_1. \tag{3}$$

Here, $X_1 \in \mathbb{R}^{b\times N}$ and $X_2 \in \mathbb{R}^{(w+b-1)\times b}$ are intermediate matrices. Since $N \gg b$ and $N \gg w$ for large-scale matrices, the primary costs of updating eigenvectors lie in the operations (1) and (3), and the operation (2) can be neglected in the following analysis.

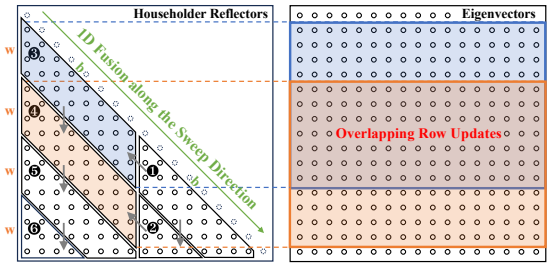

Figure 3: 1D Fusion for tridiagonal-to-band back transformation with the matrix size $N = 17$, the matrix bandwidth $w = 4$, and the block size $b = 8$. Householder reflectors generated during the band-to-tridiagonal reduction are fused into six 1D blocks along the sweep direction. The overlapping shadow represents the overlapping row updates of eigenvectors between the blue and yellow 1D blocks. The computational cost increases with b due to more overlapping row updates of eigenvectors. The gray arrows indicate the update dependencies of 1D blocks.

Number of Floating-Point Operations. The sizes of the GEMM in operations (1) and (3) are $b \times (w + b - 1) \times N$ and $(w + b - 1) \times b \times N$, respectively. The number of floating point operations for operation (1) is $2b(w + b)N$, and the same applies for operation (3). Considering there are approximately $\frac{N^2}{2wb}$ 1D blocks, the overall computational cost can be estimated as

$$Comp(b) = 4b(w + b)N \times \frac{N^2}{2wb} = 2\left(1 + \frac{b}{w}\right)N^3, \tag{4}$$

which is consistent with previous work [16].

Memory Traffic. The rows of the eigenvector E_{rows} and the matrix X_1 in operations (1) and (3) need to be read from or written to memory (e.g., global memory on the GPU) because the GEMM sizes are too large to fit into the cache. For operation (1), it is necessary to read E_{rows} and write X_1, so the memory traffic is $s(w+2b)N$, where s is the size of the data type in bytes. For operation (3), it is necessary to read X_1, and read and write E_{rows}, so the memory traffic is $s(2w+3b)N$. The memory access for X_2 is negligible because it does not involve N. Additionally, operations (1) and (3) are performed sequentially by independent GEMM operations, which means no date reuse between kernels. Therefore, the overall memory traffic of *st2sb* with 1D Fusion in bytes is:

$$Mem(b) = s(3w+5b)N \times \frac{N^2}{2wb} = s\left(\frac{3}{2b}+\frac{5}{2w}\right)N^3, \quad (5)$$

where s is the size of the data type in bytes.

Operational Intensity. Considering the number of floating-point operations and the memory traffic, *st2sb*'s OI with 1D Fusion is:

$$OI(b) = \frac{Comp(b)}{Mem(b)} = \frac{4(w+b)b}{s(3w+5b)}. \quad (6)$$

Given w, it can be shown that the derivative of OI with respect to b is positive. Therefore, $OI(b)$ increases monotonically with b.

Performance Model. The roofline model [47] can be used to estimate the lower bound of *st2sb*'s execution time with 1D Fusion:

$$T_{LB}(b) = \begin{cases} \frac{s\left(\frac{3}{2b}+\frac{5}{2w}\right)N^3}{B_{peak}}, & \text{if } OI(b) < P_{peak}/B_{peak}, \\ \frac{2\left(1+\frac{b}{w}\right)N^3}{P_{peak}}, & \text{else} \end{cases} \quad (7)$$

where B_{peak} and P_{peak} represent the peak memory bandwidth and peak performance of the platform, respectively.

This time lower bound depends on both the matrix bandwidth w and the block size b. Generally, increasing w tightens the time lower bound of *st2sb*. However, an excessively large w can prevent data from fitting into the cache during *sb2st*, significantly degrading overall EVD performance. The optimal w to minimize the end-to-end EVD solution time can be regarded as a small constant (e.g., 256) limited by the size of the private cache, such as the L2 cache on CPUs or the Local Data Memory (LDM) on the SW26010-Pro. Thus, for *st2sb* with 1D Fusion, we primarily focus on the impact of the block size b on a given platform.

When $OI(b)$ is smaller than the machine balance P_{peak}/B_{peak}, *st2sb* with 1D Fusion is memory-bound. And increasing the block size b reduces memory access volume as indicated by Eq. (5), thereby reducing $T_{LB}(b)$ in Eq. (7). However, once $OI(b)$ reaches the machine balance, further increasing b becomes futile, as it only increases the computational workload, while the performance has already reached its peak. Therefore, the optimal block size b^* that achieves the best time lower bound satisfies:

$$OI(b^*) = \frac{4(w+b^*)b^*}{s(3w+5b^*)} = P_{peak}/B_{peak}. \quad (8)$$

And the best time lower bound is:

$$T_{BLB} = \frac{2\left(1+\frac{b^*}{w}\right)N^3}{P_{peak}}. \quad (9)$$

Dongarra et al. [16] proposed an empirical recommendation of $b < 0.25w$ to optimize the execution time of *st2sb* with 1D Fusion. Nonetheless, we found this recommendation no longer suitable for modern heterogeneous architectures, where the optimal block size b^* can significantly exceed w due to the increasing machine balance. This also implies that when the OI of *st2sb* matches the machine balance required for optimal performance, the corresponding computational cost $Comp(b^*)$ in Eq. (4) may substantially exceed $4N^3$ on modern platforms, as illustrated in Figure 1 in Section 1. This trade-off between OI and computational cost makes *st2sb* a critical performance bottleneck in two-stage EVD. To further reduce the execution time of *st2sb*, it is therefore crucial to develop a new approach that achieves both high OI and low computational cost in *st2sb* on modern heterogeneous architectures. This motivates us to go beyond merely increasing the block size and to propose the 2D Fusion scheme, which decouples the OI from the computational cost of *st2sb*.

3 The 2D Fusion Scheme for Tridiagonal-to-Band Back Transformation

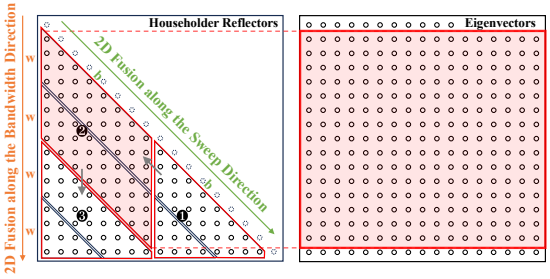

Figure 4: 2D Fusion for tridiagonal-to-band back transformation for the same example in Figure 3 with the fusion factor $l = 2$. 2D Fusion fuse every l 1D blocks in the bandwidth direction into one 2D block. The gray arrows indicate the update dependencies of three 2D blocks. The red filled 2D block is fused by the blue and yellow 1D blocks in Figure 3.

As shown in Figure 4, we observe that 1D blocks can be further fused along the bandwidth direction into **2D blocks** for updating eigenvectors, while still respecting the update dependencies. We refer to this new fusion strategy as **2D Fusion**. Compared to *st2sb* with 1D Fusion, this approach significantly increases the "effective" matrix bandwidth from w to lw, thereby further enhancing the OI of *st2sb*.

The remaining challenge is to effectively control the overall computational cost of *st2sb* with 2D Fusion. The key lies in how to utilize 2D blocks for updating eigenvectors. Suppose the rank of the blocked Householder matrix of a 2D block (the 2D-fused Householder matrix) is r, $b \le r \le lb$. The compact WY form of the 2D-fused Householder matrix becomes $H_{2D} = I + WTW^T$, where $W \in \mathbb{R}^{(lw+b-1)\times r}$. And the overall computational cost of *st2sb* is given by $2\left(\frac{r}{b} + \frac{r}{lw}\right)N^3$ with the same eigenvector update strategy as in 1D Fusion. Unfortunately, the rank r is often lb, resulting in a prohibitively high computational cost $2(l + \frac{b}{w})N^3$ for large l. However, *we find that the computational cost of st2sb with 2D Fusion is manageable when* ***explicitly constructing the 2D-fused Householder matrices*** *for updating eigenvectors*, as described in the following algorithm.

3.1 2D Fusion Algorithm

Algorithm 1 2D Fusion Back Transformation

1: **Input:** Eigenvectors of the tridiagonal matrix Q of size $n \times n$, zeroed buffers H_i, H_{f1}, H_{f2} of size $(lw + b - 1) \times (lw + b - 1)$, the 2D Fusion factor l, the block size b, and the matrix bandwidth w.
2: **Output:** The updated eigenvectors Q
3: W, T = PLARFT()
▷ Compute the compact WY representations of 1D blocks in parallel.
4: **for** each 2D block B_k in the update order (starting at the j-th row) **do**
5: LASET(H_{f1}, I)
6: LASET(H_{f2}, I)
7: $H_{f1}[0 : w + b - 1, 0 : w + b - 1]$ = LARFB($W_{k,0}$, $T_{k,0}$)
8: **for** $i = 1$ **to** $l - 1$ **do**
9: $m_1 = iw$, $m_2 = b - 1$, $m_3 = w$
10: $m_f = m_1 + m_2 + m_3$
11: LASET($H_i[0 : m_f, 0 : m_f]$), I)
12: $H_i[m_1 : m_f, m_1 : m_f]$ = LARFB($W_{k,i}$, $T_{k,i}$)
▷ Explicitly generate the Householder matrix
13: $H_{f2}[0 : m_f, 0 : m_f] = H_i \times H_{f1}$
▷ Directly multiply H_i and H_{f1}
14: swap H_{f1} and H_{f2}
15: **end for**
16: LACPY($Q[j : j + lw + b - 1, 0 : n]$,$Q_{copy}$)
▷ Copy rows of eigenvectors to a buffer
17: $Q[j : j + lw + b - 1, 0 : n] = H_{f1} \times Q_{copy}$
▷ Update eigenvectors using GEMM
18: **end for**

The 2D Fusion algorithm for *st2sb* consists of the following three fundamental building blocks:

1. Compute WY. The compact WY representations of 1D-fused Householder matrices are computed in parallel, as shown in line 3 of Algorithm 1.

2. 2D Fusion. To explicitly construct the 2D-fused Householder matrices, we first explicitly generate the 1D-fused Householder matrices and then fuse them with the correct offsets in the bandwidth direction by GEMM. To be specific, consider the i-th 1D block $B_{k,i}$ of the k-th 2D block B_k in the update order. The 1D-fused Householder matrix of $B_{k,i}$ is constructed on-the-fly using LARFB, which calculates $W_{k,i}T_{k,i}W_{k,i}^T \in \mathbb{R}^{(w+b-1)\times(w+b-1)}$ in two GEMM operations and performs a matrix addition $I+W_{k,i}T_{k,i}W_{k,i}^T$. Due to the varying starting rows of 1D blocks, the buffer H_i is set to an identity matrix by LASET in advance, and then the constructed 1D-fused Householder matrix $I + W_{k,i}T_{k,i}W_{k,i}^T$ is placed as a diagonal submatrix of H_i with an offset matching the starting row of $B_{k,i}$, as shown in lines 11 and 12 of Algorithm 1. Finally, the matrix product $H_i \times H_{f1}$ is computed to fuse $B_{k,i}$, where H_{f1} is the 2D-fused Householder matrix from the first $i-1$ 1D blocks, as shown in line 13 of Algorithm 1. After $l-1$ such iterations, the 2D-fused Householder matrix of B_k is stored in H_{f1}.

3. Row updates of Eigenvectors. To update the relevant rows of the eigenvectors Q for each 2D block, additional row copies of the eigenvectors (LACPY) are required due to the lack of in-place support for GEMM operations. Then the copied rows of eigenvectors Q_{copy} are multiplied by the 2D-fused Householder matrix H_{f1} from the left and stored back to eigenvectors Q.

3.2 Analytical Performance Model

To demonstrate the superiority of 2D Fusion over the conventional 1D Fusion, we present an analytical performance model in this section. The model captures the cost of updating eigenvectors using the LACPY and GEMM kernels, and also analyzes the intrinsic overhead of explicitly constructing 2D-fused Householder matrices.

❶ ***GEMM for updating eigenvectors.*** The GEMM size in line 17 of Algorithm 1 is $(lw + b - 1) \times (lw + b - 1) \times N$. Since there are approximately $\frac{N^2}{2lwb}$ 2D blocks, the overall computational cost can be estimated as

$$2(lw + b)^2 N \times \frac{N^2}{2lwb} = \left(2 + \frac{b}{lw} + \frac{lw}{b}\right) N^3. \tag{10}$$

For each GEMM operation, the input matrix Q_{copy} is read, and the matrix product $H_{f1} \times Q_{copy}$ is written back to the corresponding $(lw+b-1)$ rows of the eigenvectors Q. Since the memory access for H_{f1} is negligible when $lw + b \ll N$, the total memory traffic of GEMM operations in bytes is approximated as

$$2s(lw + b)N \times \frac{N^2}{2lwb} = s\left(\frac{1}{b} + \frac{1}{lw}\right) N^3, \tag{11}$$

where s denotes the size of the data type in bytes. So the OI of GEMM for updating eigenvectors is:

$$OI(l, b) = \frac{lw + b}{s}. \tag{12}$$

❷ ***LACPY for row copies of eigenvectors.*** For each 2D block, the row copies of eigenvectors in line 16 of Algorithm 1 read $(lw + b - 1)$ rows of the eigenvectors Q and store them in the buffer Q_{copy}. The corresponding memory traffic is:

$$2s(lw + b)N \times \frac{N^2}{2lwb} = s\left(\frac{1}{b} + \frac{1}{lw}\right)N^3, \tag{13}$$

❸ ***Intrinsic cost of 2D Fusion.*** The 2D Fusion cost consists of two main parts:

(1) Cost of explicitly generating the 1D-fused Householder matrices by the LARFB *kernel.* It primarily involves two GEMM operations of size $(w+b-1)\times b\times b$ and $(w+b-1)\times b\times(w+b-1)$, respectively. The total number of floating-point operations required to construct the $\frac{N^2}{2wb}$ 1D-fused Householder matrices is listed in Table 3.

(2) Cost of calculating the 2D-fused Householder matrices. As shown in Table 3, the memory write overhead is introduced by LASET, which initializes the buffer H_i as an identity matrix. The floating-point computational overhead arises from using GEMM operations to fuse the 1D-fused Householder matrices. After each fusion step, the size of the intermediate 2D-fused Householder matrices increases by w, leading to the summation form in Table 3.

Table 3: Overhead of 2D Fusion. w denotes the matrix bandwidth, b represents the block size of 2D Fusion, l is the fusion factor, and H_i is the 1D-fused Householder matrix. GEMM accounts for floating-point operations, while the matrix initialization LASET accounts for memory operations.

Operation	Total Mem./FP ops.	Kernel
Explicit H_i generation	$N^2\frac{(w+b)(w+2b)}{w}$	LARFB(GEMM)
2D Fusion of H_i	$N^2\frac{\sum_{k=2}^{l}(kw+b)^2}{2lwb}$	LASET
	$N^2\frac{\sum_{k=2}^{l}(kw+b)^3}{lwb}$	GEMM

Performance Model. The lower bound of the execution time of *st2sb* with 2D Fusion can be estimated as

$$T_{LB}(l,b) = T_{update}(l,b) + C \tag{14}$$

$$= T_{GEMM}(l,b) + T_{LACPY}(l,b) + C \tag{15}$$

$$= \frac{\left(2 + \frac{b}{lw} + \frac{lw}{b}\right)N^3}{P_{peak}} + \frac{s\left(\frac{1}{b} + \frac{1}{lw}\right)N^3}{B_{peak}} + C. \tag{16}$$

where C is the cost of 2D Fusion, and P_{peak} and B_{peak} denote the peak computational performance and peak memory bandwidth of the platform, respectively.

By maintaining $lw \approx b$, the computational cost of Eq. (10) can be controlled to approach its minimum value of $4N^3$. Furthermore, by proportionally increasing both lw and b, we can flexibly improve the size and OI of GEMM until its performance saturates near P_{peak}, as shown in Eq. (12). At the same time, the memory traffic of LACPY is proportionally reduced. Moreover, LACPY can easily reach a bandwidth close to B_{peak}, since it only consists of memory copy operations, such as device-to-device transfers on a GPU. Consequently, *st2sb* with 2D Fusion can outperform its 1D Fusion counterpart on architectures with high machine balance, provided that C is well controlled. We observe that when the fusion factor l becomes large, the 2D Fusion cost C can offset the benefits of 2D Fusion in terms of both reduced computation and increased OI over 1D Fusion for updating eigenvectors. To mitigate this issue, we introduce a skipping strategy to control the overhead of 2D Fusion for large fusion factors in the next section.

3.3 2D Fusion Algorithm with Skipping

We identify that when the fusion factor l is large, the substantial overhead of 2D Fusion in Table 3 stems from unnecessary operations involving the identity matrix in the upper-left corner of H_i when computing $H_i \times H_{f1}$. As 2D Fusion progresses, the dimension of this identity matrix, denoted as m_1 in Algorithm 1, increases by w in each fusion step. As the fusion factor l becomes large, the number of fusion steps increases proportionally, resulting in growing unnecessary computations that substantially raise the cost of 2D Fusion and should thus be avoided.

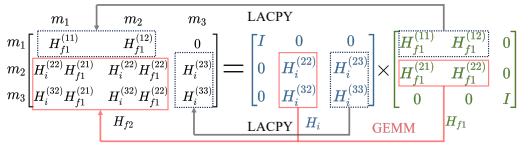

Figure 5: Computing the intermediate 2D-fused Householder matrix H_{f2} with the skipping strategy. The computations with identity matrices can be skipped. m_1 increases progressively by w during the 2D Fusion process, while $m_2 = b - 1$ and $m_3 = w$.

To address this issue, we propose a skipping strategy, as illustrated in Figure 5. Our skipping strategy employs block matrix multiplications to compute the lower-left part of the intermediate 2D-fused Householder matrix H_{f2} without involving operations with identity matrices, which significantly reduces the computational cost of 2D Fusion for large fusion factors. The original multiplications with identity matrices are replaced with memory copies to construct the remaining parts of H_{f2}, which can be efficiently performed using LACPY. Compared to the original 2D Fusion algorithm, the need to explicitly initialize buffers as identity matrices by LASET is also eliminated, as our skipping strategy does not

require access to identity matrices. The detailed procedure of the 2D Fusion algorithm with skipping is presented in Algorithm 2, with the optimized 2D Fusion cost summarized in Table 4.

Algorithm 2 2D Fusion Back Transformation with Skipping

Input: Eigenvectors of the tridiagonal matrix Q of size $n \times n$, zeroed buffers H_i, H_{f1}, H_{f2} of size $(lw + b - 1) \times (lw + b - 1)$, the 2D Fusion factor l, the block size b, and the matrix bandwidth w.
Output: The updated eigenvectors Q
W, T = PLARFT()
▷ Compute the compact WY representations of 1D blocks in parallel.
for each 2D block B_k in the update order (starting at the j-th row) **do**
  $H_{f1}[0 : w + b - 1, 0 : w + b - 1]$ = LARFB($W_{k,0}, T_{k,0}$)
  for $i = 1$ **to** $l - 1$ **do**
    $m_1 = iw, m_2 = b - 1, m_3 = w$
    $m_f = m_1 + m_2 + m_3$
    $H_i[m_1 : m_f, m_1 : m_f]$ = LARFB($W_{k,i}, T_{k,i}$)
    ▷ Explicitly generate the Householder matrix
    copy the $m_1 \times (m_1 + m_2)$ submatrix of H_{f1} to H_{f2}
    copy the $(m_2 + m_3) \times m_3$ submatrix of H_i to H_{f2}
    update H_{f2} using $BLOCK\_GEMM(H_i, H_{f1})$
    swap H_{f1} and H_{f2}
  end for
  LACPY($Q[j : j + lw + b - 1, 0 : n]$,$Q_{copy}$)
  ▷ Copy rows of eigenvectors to a buffer
  $Q[j : j + lw + b - 1, 0 : n] = H_{f1} \times Q_{copy}$
  ▷ Update eigenvectors using GEMM
end for

Table 4: Overhead of 2D Fusion with skipping. w denotes the matrix bandwidth, b represents the block size of 2D Fusion, l is the fusion factor, and H_i is the 1D-fused Householder matrix. GEMM accounts for floating-point operations, while the matrix copying LACPY accounts for memory operations.

Operation	Total Mem./FP ops.	Kernel
Explicit H_i generation	$N^2 \frac{(w+b)(w+2b)}{w}$	LARFB(GEMM)
2D Fusion of H_i with skipping	$N^2((\frac{l^2}{3}+1)\frac{w}{b}+\frac{l}{2})$	LACPY
	$N^2 \frac{l(w+b)(lw+2b)}{2(l-1)w}$	GEMM

3.4 Platform-Specific Optimizations

We implemented our 2D Fusion method in a library called Eigen2D, based on existing two-stage eigensolvers in MAGMA and LAPACK, targeting both NVIDIA GPUs and the SW26010-Pro processor. Due to the differences in computational and memory access characteristics across these platforms, we additionally introduced the following platform-specific optimizations for 2D Fusion.

Memory Alignment for SGEMM with TF32 on NVIDIA GPUs. We observed that memory alignment significantly impacts the performance of TF32-accelerated SGEMM on NVIDIA GPUs, with the best performance achieved when the alignment is 256-bit. To optimize the row updates of eigenvectors, we align both H_{f1} and Q_{copy} to 256-bit. To ensure proper alignment of the starting rows of eigenvectors Q for every 2D-fused Householder matrix, the following three conditions must be satisfied: (1) $Q + 1$ must be 256-bit aligned to ensure that the starting row of eigenvectors corresponding to the upper-left 2D block is 256-bit aligned; (2) The block size b should be a multiple of 8 to ensure that the starting row of eigenvectors for the first 2D block in each column remains 256-bit aligned if condition (1) is satisfied; (3) lw should be a multiple of 8 to guarantee that the starting row of eigenvectors corresponding to subsequent 2D blocks in the same column remains 256-bit aligned, provided that condition (2) is satisfied. Furthermore, the size of H_{f1} is padded to the nearest power of 2 to enhance GEMM performance.

Optimizations for Memory Access on the SW26010-Pro. Due to the constraints of memory bandwidth and LDM size on the SW26010-Pro, we propose the following two strategies to optimize memory access for *st2sb* with 2D Fusion: (1) To reduce the number of memory accesses required for computing the compact WY representation, we developed a compact version of the PLARFT function by eliminating unnecessary GEMV operations during the computation of matrix T. On CPU-GPU heterogeneous architectures, the benefits of this compact version are negligible due to their significantly higher memory bandwidth and more efficient cache utilization compared to the SW26010-Pro; (2) To further enhance the performance of the LACPY function in xMath 2.0 [30], the SOTA mathematical library on the SW26010-Pro, we implemented an optimized LACPY version using DMA, which is specifically designed for efficient transfer of large data blocks.

4 Evaluation

4.1 Experimental Setup

Platforms. We selected two platforms to evaluate the performance of Eigen2D on modern heterogeneous architectures. The first is a CPU-GPU heterogeneous system equipped with NVIDIA A100 GPUs, which have a machine balance close to 100 under TensorFloat-32 (TF32) precision [37]. We refer to this platform as System A100. The second platform is the next-generation Sunway supercomputer, equipped with SW26010-Pro processors with a machine balance of approximately 50 for both double-precision (FP64) and single-precision (FP32) computations. The hardware and software configurations of these two platforms are summarized in Table 5.

Baselines. We use the two-stage eigensolvers with 1D Fusion from the latest release of **MAGMA v2.8.0** [44], magma_dsyevdx_2stage and magma_ssyevdx_2stage, as our baselines on System A100. These solvers are optimized for heterogeneous platforms with multi-core CPUs and NVIDIA

Table 5: Hardware and software configurations

Platform	System A100	Next-generation Sunway
Processor	CPU: 2×AMD EPYC 7742 GPU: 8×NVIDIA A100-PCIE	SW26010-Pro 6 Core Groups (CGs) 1 MPE + 64 CPEs per CG
Memory	CPU: 512 GB DDR4 GPU: 40 GB HBM2	16 GB per CG
Peak Memory Bandwidth	GPU: 1.56 TB/s	51.2 GB/s per CG
Cache	CPU: 32 KB L1d cache 512 KB L2 cache	256 KB LDM per CPE
Peak Performance	FP64 19.5 TFLOP/s FP32 19.5 TFLOP/s TF32 155.9 TFLOP/s	FP64 2.3 TFLOP/s per CG FP32 2.3 TFLOP/s per CG
Libraries	Intel MKL `2021.4.0` CUDA `12.4.1` cuSOLVER `11.6.1` cuBLAS `12.4.5`	xMath 2.0 for BLAS and LAPACK
Compilers	Intel `icpx 2021.4.0` NVIDIA `nvcc 12.4.131`	`swgcc 7.1.0`

GPUs. As a naive mixed-precision scheme, we additionally configured the compute type of `cublasGemmEx` to `CUBLAS_COMPUTE_32F_FAST_TF32` to support TF32 during *st2sb*, while all other EVD routines were performed in FP32. On the next-generation Sunway supercomputer, we ported `dsyevd_2stage` and `ssyevd_2stage` from the latest **LAPACK 3.12.0** [2] to the SW26010-Pro by implementing and tuning both *sb2st* and *st2sb* with 1D Fusion on CPEs, which serve as our baselines. **Our Eigen2D with 2D Fusion is based on two-stage eigensolvers in MAGMA and LAPACK, incorporating the same optimizations for all EVD routines except *st2sb*.**

Design of Experiments. We demonstrate that Eigen2D outperforms the SOTA eigensolvers with 1D Fusion in *st2sb* through a detailed analysis of execution time. Additionally, we validate the effectiveness of our skipping strategy and platform-specific optimizations. In the end-to-end performance evaluation, we also introduce SOTA one-stage eigensolvers as additional baselines: (1) `cusolverDnXsyevd` from cuSOLVER [35] on System A100 (matrices larger than 28k are not supported due to a `cusolverDnXsyevd_bufferSize` error); and (2) `dsyevd` and `ssyevd` from xMath 2.0 [30], the SOTA mathematical library on the SW26010-Pro. The experimental results highlight the superiority of Eigen2D over other eigensolvers for large-scale matrices.

On System A100, we conducted experiments with a single GPU and a single CPU. The matrix sizes range from 4096 to 40960. On the SW26010-Pro, all tests were conducted using a single core group (1 MPE + 64 CPEs), with matrix sizes varying from 1024 to 10240. The test matrices for performance evaluation on both platforms were randomly generated symmetric matrices, as the eigenvalue distribution has minimal impact on the performance of direct eigensolvers based on tridiagonalization. And all eigenvalue/vector pairs are solved.

Both 1D Fusion and 2D Fusion require parameter tuning for minimize the end-to-end EVD runtime. So we performed an extensive parameter search for two-stage eigensolvers using the largest matrix size in our performance test. On System A100, we explored matrix bandwidths w ranging from 64 to 512 with an increment of 32. For 1D Fusion, the block size b is varied from $w/4$ to $2w$ in increments of $w/4$, covering the parameter space suggested in existing two-stage solvers [16, 26, 45]. For 2D Fusion, we used the same range of w with $l = 1, 2, 3, 4$ and $b = lw$. On SW26010-Pro, w ranged from 64 to 256 in increments of 32. We tuned $w + b - 1$ for 1D Fusion and $lw + b - 1$ for 2D Fusion from 128 to 2048, ensuring that it remained a power of two to optimize GEMM performance on the SW26010-Pro. The optimal parameters are summarized in Table 6.

Table 6: The searched optimal parameters for two-stage eigensolvers with 1D Fusion and 2D Fusion. "Comp" and "OI" represent the computational cost and operational intensity of *st2sb*, respectively.

Platform	Precision	Fusion	w	b	l	Comp	OI
System A100	FP64	1D	128	64	-	$3.0N^3$	8.7
		2D	128	256	2	$4.0N^3$	32.0
	FP32	1D	128	64	-	$3.0N^3$	17.5
		2D	128	256	2	$4.0N^3$	64.0
	TF32	1D	256	256	-	$4.0N^3$	64.0
		2D	128	384	3	$4.0N^3$	96.0
Next-generation Sunway	FP64	1D	96	417	-	$10.7N^3$	45.0
		2D	96	353	7	$4.4N^3$	64.1
	FP32	1D	128	385	-	$8.0N^3$	85.5
		2D	96	353	7	$4.4N^3$	128.1

Accuracy Evaluation. We generated test matrices of size 2048 for accuracy evaluation by the standard `LATMS` routine [3], consistent with LAPACK, PLASMA, and MAGMA. These matrices were constructed as $A = Q_0\Sigma_0Q_0^T$ with condition numbers ranging from 10^2 to 10^{20}, where the eigenvectors Q_0 are random orthogonal matrices, and Σ_0 represents the generated eigenvalues. The eigenvalue distributions include "Arithmetic" and "Geometric". Definitions of these distributions can be found in [16]. To evaluate the accuracy of eigensolvers, the error $\frac{||A-Q\Sigma Q^T||_1}{\epsilon N||A||_1}$ is computed using the `SYT21` routine from LAPACK [3], where Σ and Q are the computed eigenvalues and eigenvectors, respectively, and $||\cdot||_1$ denotes the 1-norm.

4.2 Performance of Tridiagonal-to-Band Back Transformation

Figure 6 compares the execution time of *st2sb* in Eigen2D with SOTA two-stage eigensolvers using 1D Fusion. On System A100, for matrix sizes smaller than 24k, the performance of Eigen2D is constrained by the overhead introduced by 2D Fusion and the computation of the compact WY representation with a larger block size b compared to MAGMA.

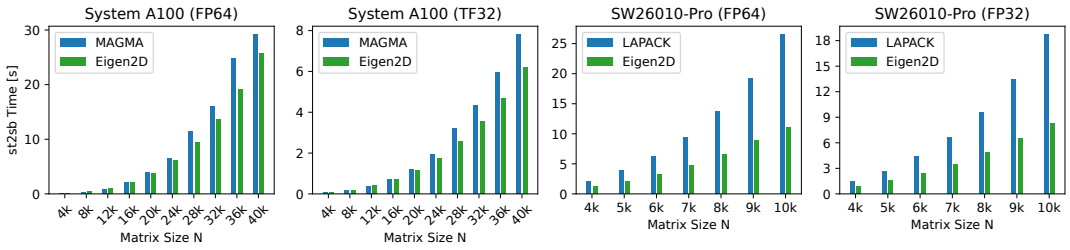

Figure 6: Time comparison of the tridiagonal-to-band back transformation *st2sb*. Eigen2D employs 2D Fusion, whereas MAGMA and LAPACK utilize 1D Fusion. The FP32 results on System A100 are explained in Section 4.2.

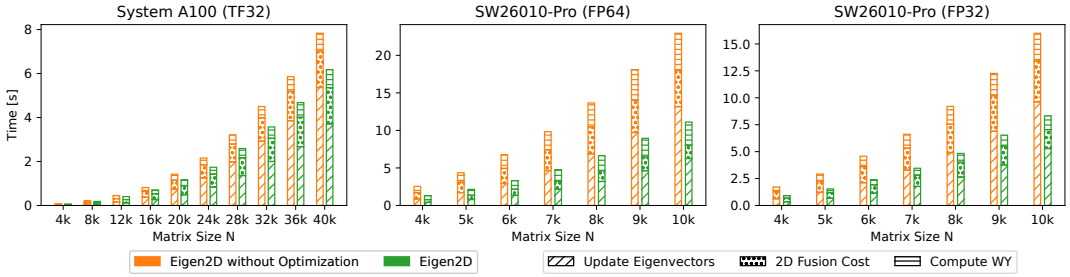

Figure 7: Optimization effects of skipping and platform-specific optimizations for *st2sb* in Eigen2D.

As the matrix size increases, the performance advantage of Eigen2D becomes more pronounced, achieving an average of 1.21× and 1.25× speedups over MAGMA for matrix sizes larger than 24k in FP64 and TF32, respectively. Take the TF32 version as an example, the speedup of Eigen2D over MAGMA primarily stems from a 3.06× average speedup in GEMM performance, benefiting from enhanced OI and our optimizations of memory alignment for SGEMM with TF32 on the Nvidia A100 GPU. The overhead of the LACPY kernel is negligible, as it achieves over 70% of the theoretical memory bandwidth of the NVIDIA A100 GPU for large matrices, with only $N^3/48$ memory traffic. For FP32 precision (omitted from Figure 6), Eigen2D performs comparably to MAGMA. Due to the low machine balance in FP32, *st2sb* with 1D Fusion in MAGMA already achieves near-saturated performance under the configuration in Table 6. In this case, the benefit of further improving OI by 2D Fusion cannot offset the 25% increase in computations and the additional 2D Fusion cost.

On the SW26010-Pro, Eigen2D significantly outperforms the two-stage eigensolvers of LAPACK, achieving an average speedup of 1.97× in FP32 and 1.91× in FP64. This performance gain is driven by two factors. First, Eigen2D exhibits higher OI and superior GEMM performance compared to LAPACK's two-stage eigensolvers. Second, the computational cost of *st2sb* is significantly reduced by a factor of 1.82× in FP32 and 2.43× in FP64 in Eigen2D as shown in Table 6. Compared to System A100, Eigen2D performs better on the SW26010-Pro primarily due to its higher machine balance (50 for FP64/FP32) versus that of Nvidia A100 GPUs (12.5 for FP64/FP32). Moreover, the Sunway architecture suffers from coarse-grained memory access, which prevents GEMM performance from reaching the roofline. This limitation further amplifies the advantage of 2D Fusion in improving OI on the SW26010-Pro.

Figure 7 illustrates the effectiveness of the skipping strategy and platform-specific optimizations in Eigen2D. On System A100, the effect of skipping under the simple mixed-precision scheme is limited due to the small fusion factor in Eigen2D, where the primary fusion overhead arises from explicitly generating Householder matrices rather than from the GEMM operations for 2D Fusion. A similar phenomenon is also observed in both FP32 and FP64 precision. In contrast, memory alignment for SGEMM with TF32 plays a crucial role in enhancing the performance of eigenvector updates, leading to an average of 1.63× speedup. On the SW26010-Pro, the skipping strategy proves to be highly effective, reducing fusion overhead by 49.0% in FP32 and 59.8% in FP64. Additionally, the compact version of PLARFT mitigates redundant memory access, shortening execution time to 50.2% and 62.6% of the original in single and double precision, respectively. Furthermore, the optimized LACPY significantly accelerates the eigenvector copying process, achieving 8.01× and 6.80× higher memory bandwidth compared to LACPY in xMath 2.0 for single and double precision, respectively.

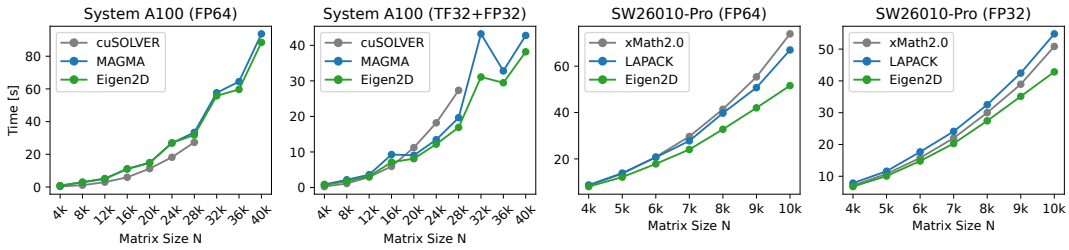

Figure 8: End-to-end eigenvalue decomposition time on System A100 and SW26010-Pro.

These results highlight the effectiveness of our 2D Fusion optimizations in enhancing kernel performance and reducing both computational cost and memory access overhead.

4.3 End-to-End Performance

Figure 8 presents the end-to-end solution time for eigenvalue decomposition. For FP64 on System A100, Eigen2D achieves an average speedup of 1.06× over MAGMA for matrix sizes larger than 24k, while the acceleration is minimal for smaller matrices due to the intrinsic cost of 2D Fusion. In our simple TF32+FP32 mixed-precision scheme, Eigen2D achieves up to a 1.30× speedup and an average speedup of 1.17× compared to MAGMA. Note that the 32k case is excluded from this average, as the performance of *sb2sy* degrades significantly due to the anomalously poor efficiency of Intel MKL when computing the compact WY representation on the CPU. The performance of Eigen2D is further enhanced by its use of a smaller matrix bandwidth w than MAGMA as shown in Table 6, which reduces memory accesses and improves data locality in *sb2st*. The overall EVD performance of Eigen2D in FP32 is comparable to MAGMA (not included in Figure 8).

On SW26010-Pro, Eigen2D consistently outperforms other eigensolvers across all matrix sizes, primarily due to its significantly improved `GEMM` performance in *st2sb*. Compared to the two-stage eigensolvers in LAPACK, it achieves an average speedup of 1.19× in FP32 and 1.18× in FP64. Overall, 2D Fusion proves to be an encouraging approach for solving eigensystems of large-scale symmetric matrices on modern heterogeneous architectures.

4.4 Accuracy

As illustrated in Figure 9, the errors of two-stage eigensolvers with 1D and 2D fusion are nearly identical in both FP64 and FP32, ranging from 10^{-2} to 10^{-1}. These results are comparable to those of one-stage eigensolvers and align with the expectations in [3]. In our simple TF32+FP32 mixed-precision scheme, both 1D Fusion and 2D Fusion exhibit a three-order-of-magnitude increase in error compared to the FP32 eigensolvers, primarily due to mantissa truncation during back transformation of eigenvectors in *st2sb*. This issue could potentially be mitigated using techniques such as iterative refinement [43]. Nevertheless, 2D Fusion still achieves higher accuracy than 1D Fusion in the TF32+FP32 case, benefiting from the use of FP32 for constructing 2D-fused Householder matrices instead of TF32. Although a thorough investigation of mixed-precision EVD with iterative refinement is beyond the scope of this paper, we believe that the performance advantage of 2D Fusion over 1D Fusion would still hold in such cases.

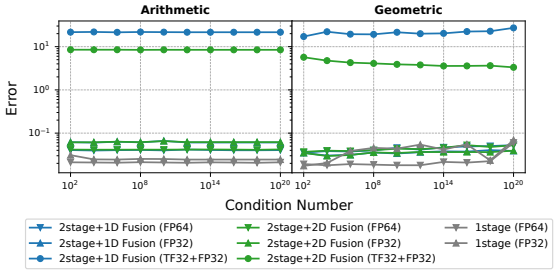

Figure 9: EVD Errors $\frac{||A-Q\Sigma Q^T||_1}{\epsilon N||A||_1}$ for matrices of size 2048 using one-stage and two-stage eigensolvers with FP64, FP32, and TF32+FP32. "Arithmetic" and "Geometric" refer to the eigenvalue distributions.

5 Related Work

The calculation of eigenvalues and eigenvectors remains a cornerstone in both numerical linear algebra and real-world applications [4, 16, 26, 28, 32, 49]. Methods for solving eigensystems of symmetric matrices include subspace iteration methods [29, 40], Jacobi methods [5, 7], the one-stage tridiagonalization method [2] and the two-stage tridiagonalization method [10].

Subspace iteration methods, such as the Power Iteration [13], the Lanczos [38] method, and the Jacobi-Davidson method [40], compute eigenvalues and eigenvectors by iteratively refining an initial solution in a low-dimensional subspace (e.g., Krylov subspaces [29]). These methods are efficient for finding extreme eigenvalues in large-scale sparse

or structured matrices. However, they are not suitable when a large portion of the eigensystem is required, as they rely on memory-bound matrix-vector multiplications using BLAS-2 kernels [12], which can limit performance.

Jacobi methods [14] are simple, accurate, and highly parallelizable. They compute the eigenvalues and corresponding eigenvectors by transforming the matrix into a diagonal form through a series of sweeps of Givens rotations [8]. To improve OI, the Block Jacobi method [7] can be used. And preconditioning and parallel orderings are employed to accelerate convergence [5, 6, 19, 34]. However, for matrices of size larger than 1024, Jacobi methods are generally impractical due to their high computational complexity of $O(SN^3)$, where S, the number of sweeps required for convergence, is typically greater than 10.

The one-stage tridiagonalization method directly reduces the original matrix to a tridiagonal form using blocked Householder transformations. However, the one-stage reduction is memory-bound because half of the computational workload is performed by BLAS-2 `SYMV` operations. This becomes the primary performance bottleneck that prevents the one-stage eigensolvers from achieving satisfactory efficiency for large-scale matrices, such as `dsyevd` in LAPACK [2] and Intel MKL [27], and `cusolverDnSsyevd` in CuSOLVER [35]. Recently, EigenExa [20] advanced one-stage eigensolvers for petascale applications with a novel approach: a narrow-band reduction followed by a band divide-and-conquer [23] method, which still requires further optimization to enhance performance and scalability.

The two-stage tridiagonalization method was proposed by Bischof et al. [9] as a solution to the memory bottleneck of the one-stage tridiagonalization, by introducing an additional band reduction step. As the scale of eigenproblems continues to grow, this approach has been widely adopted in high performance eigensolvers, including:

The SBR toolbox on CPUs [11]. To improve the efficiency of the tridiagonal-to-band back transformation, it employs a 1D Fusion strategy, which groups multiple Householder reflectors into a single blocked transformation. However, 1D Fusion faces a trade-off between OI and the computational cost, making the tridiagonal-to-band back transformation a performance bottleneck.

PLASMA on multi-core architectures [45]. Luszczek et al. designed an asynchronous algorithm of band-to-tridiagonal reduction for multi-core CPUs in PLASMA [31, 48]. Haidar et al. [24] expanded the PLASMA library with support for eigenvector computations through back transformation. However, PLASMA still uses the 1D Fusion strategy during tridiagonal-to-band back transformation thus inheriting the same issue as SBR.

MAGMA on CPU-GPU heterogeneous architectures [42]. MAGMA extended the two-stage eigensolvers of PLASMA to heterogeneous platforms. To handle large-scale eigenvalue problems, multi-GPU implementation is also supported. However, back transformation using 1D fusion still remains a significant performance bottleneck in MAGMA for large-scale EVD.

ELPA on distributed heterogeneous platforms [49]. Marek et al. [32] developed the ELPA library to solve large-scale eigenproblems of dense symmetric and Hermitian matrices across computing nodes. They employed a non-WY method to improve the scalability of back transformation, which directly applys Householder transformations using BLAS-2 kernels.

As the machine balance of modern heterogeneous architectures continues to increase, the SOTA two-stage EVD algorithm with 1D Fusion faces growing challenges due to the dilemma between limited OI and excessive computational cost. Our 2D Fusion method provides an effective solution by decoupling the OI and the computational cost of *st2sb*. It also has the potential to support multi-GPU implementations and cluster computing by column-wise partitioning of eigenvectors, and can be extended to a broad range of current and emerging heterogeneous architectures.

6 Conclusion and Future Work

In this work, we introduce a flexible 2D Fusion method by further fusing the blocked Householder transformations from 1D Fusion. It achieves both high OI and low computational cost in the tridiagonal-to-band back transformation for eigenvectors, which is often the performance bottleneck of the two-stage tridiagonalization EVD method. To further accelerate EVD with 2D Fusion at large fusion factors, we propose a skipping strategy that reduces the intrinsic cost of 2D Fusion by avoiding unnecessary matrix computations involving identity matrices. We then integrate 2D Fusion into SOTA two-stage eigensolvers in MAGMA and LAPACK to demonstate the effectiveness of our approach. Experimental results show that 2D Fusion significantly accelerates these SOTA two-stage eigensolvers on CPU-GPU heterogeneous architectures and the next-generation Sunway supercomputer. In future work, we plan to extend our 2D Fusion method into a mixed-precision eigensolver with iterative refinement for accuracy recovery [22, 43] to fully leverage both the OI and accuracy advantages of 2D Fusion over 1D Fusion on modern heterogeneous architectures.

Acknowledgments

This work was primarily supported by the National Key R&D Program of China (2023YFB3001900), and was also partially supported by the National Natural Science Foundation of China (No. U2242210). The corresponding author of this work is Prof. Wei Xue (xuewei@mail.tsinghua.edu.cn).

References

[1] Advanced Micro Devices, Inc. 2024. AMD matrix cores. https://rocm.blogs.amd.com/software-tools-optimization/matrix-cores/README.html Accessed: 2024-11-25.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[3] Edward Anderson, Jack J. Dongarra, and Susan Ostrouchov. 1992. LAPACK Working Note 41: Installation Guide for LAPACK. https://api.semanticscholar.org/CorpusID:16481322

[4] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P.R. Willems. 2011. Parallel Solution of Partial Symmetric Eigenvalue Problems from Electronic Structure Calculations. 37, 12 (2011), 783–794. doi:10.1016/j.parco.2011.05.002

[5] M Bečka, G Okša, and M Vajteršic. 2002. Dynamic Ordering for a Parallel Block-Jacobi SVD Algorithm. *Parallel Comput.* 28, 2 (Feb. 2002), 243–262. doi:10.1016/S0167-8191(01)00138-7

[6] Martin Bečka, Gabriel Okša, and Marián Vajteršic. 2015. New dynamic orderings for the parallel one–sided block-Jacobi SVD algorithm. *Parallel Processing Letters* 25, 02 (2015), 1550003.

[7] MARTIN BEČKA and Marián Vajteršic. 1999. Block-Jacobi SVD algorithms for distributed memory systems I: Hypercubes and rings. *Parallel Algorithms and Application* 13, 3 (1999), 265–287.

[8] David Bindel, James Demmel, William Kahan, and Osni Marques. 2002. On computing Givens rotations reliably and efficiently. *ACM Transactions on Mathematical Software (TOMS)* 28, 2 (2002), 206–238.

[9] C. Bischof, Xiaobai Sun, and B. Lang. 1994. Parallel tridiagonalization through two-step band reduction. In *Proceedings of IEEE Scalable High Performance Computing Conference*. 23–27. doi:10.1109/SHPCC.1994.296622

[10] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. 2000. Algorithm 807: The SBR Toolbox—Software for Successive Band Reduction. 26, 4 (2000), 602–616. doi:10.1145/365723.365736

[11] Christian H. Bischof, Bruno Lang, and Xiaobai Sun. 2000. Algorithm 807: The SBR Toolbox—Software for Successive Band Reduction. *ACM Trans. Math. Software* 26, 4 (Dec. 2000), 602–616. doi:10.1145/365723.365736

[12] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[13] Thomas E Booth. 2006. Power iteration method for the several largest eigenvalues and eigenfunctions. *Nuclear science and engineering* 154, 1 (2006), 48–62.

[14] James Demmel and Krešimir Veselić. 1992. Jacobi's method is more accurate than QR. *SIAM journal on matrix analysis and applications* 13, 4 (1992), 1204–1245.

[15] Inderjit S Dhillon and Beresford N Parlett. 2004. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra Appl.* 387 (2004), 1–28.

[16] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2018. The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale. *SIAM review* 60, 4 (2018), 808–865.

[17] Jack J Dongarra, Danny C Sorensen, and Sven J Hammarling. 1989. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comput. Appl. Math.* 27, 1-2 (1989), 215–227.

[18] Augustin A Dubrulle. 2000. Householder transformations revisited. *SIAM J. Matrix Anal. Appl.* 22, 1 (2000), 33–40.

[19] Patricia J. Eberlein and Haesun Park. 1990. Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures. *J. Parallel and Distrib. Comput.* 8, 4 (1990), 358–366.

[20] Takeshi Fukaya and Toshiyuki Imamura. 2015. EigenExa. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 960–969. doi:10.1109/IPDPSW.2015.128

[21] Jiangang Gao, Fang Zheng, Fengbin Qi, Yajun Ding, Hongliang Li, Hongsheng Lu, Wangquan He, Hongmei Wei, Lifeng Jin, Xin Liu, et al. 2021. Sunway supercomputer architecture towards exascale computing: analysis and practice. *Science China Information Sciences* 64, 4 (2021), 141101.

[22] Weiguo Gao, Yuxin Ma, and Meiyue Shao. 2022. A mixed precision Jacobi SVD algorithm. *arXiv preprint arXiv:2209.04626* (2022).

[23] Ming Gu and Stanley C Eisenstat. 1995. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Matrix Anal. Appl.* 16, 1 (1995), 79–92.

[24] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. 2013. An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013-11-17). ACM, 1–12. doi:10.1145/2503210.2503292

[25] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. 2013. An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 1–12. doi:10.1145/2503210.2503292

[26] A Haidar, S Tomov, I Yamazaki, R Solca, T Schulthess, T Dong, and J Dongarra. 2008. Magma: A breakthrough in solvers for eigenvalue problems.

[27] Intel Corporation. 2024. Intel MKL Library. http://software.intel.com/en-us/articles/intel-mkl/ Accessed: 2024-11-25.

[28] Yiyuan Li, Xiting Ju, Yi Xiao, Qilong Jia, Yongxiao Zhou, Simeng Qian, Rongfen Lin, Bin Yang, Shupeng Shi, Xin Liu, et al. 2023. Rapid simulations of atmospheric data assimilation of hourly-scale phenomena with modern neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[29] Jörg Liesen and Zdenek Strakos. 2013. *Krylov subspace methods: principles and analysis*. Numerical Mathematics and Scie.

[30] Fangfang Liu, Wenjing Ma, Yuwen Zhao, Daokun Chen, Yi Hu, Qinglin Lu, WanWang Yin, Xinhui Yuan, Lijuan Jiang, Hao Yan, et al. 2023. xmath2. 0: a high-performance extended math library for sw26010-pro many-core processor. *CCF Transactions on High Performance Computing* 5, 1 (2023), 56–71.

[31] Piotr Luszczek, Hatem Ltaief, and Jack Dongarra. 2011. Two-Stage Tridiagonal Reduction for Dense Symmetric Matrices Using Tile Algorithms on Multicore Architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium* (2011-05). IEEE, 944–955. doi:10.1109/IPDPS.2011.91

[32] Andreas Marek, Volker Blum, Rainer Johanni, Ville Havu, Bruno Lang, Thomas Auckenthaler, Alexander Heinecke, Hans-Joachim Bungartz, and Hermann Lederer. 2014. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Journal of Physics: Condensed Matter* 26, 21 (2014), 213201.

[33] Osni Marques and Paulo B Vasconcelos. 2017. Computing the bidiagonal SVD through an associated tridiagonal eigenproblem. In *High Performance Computing for Computational Science–VECPAR 2016: 12th International Conference, Porto, Portugal, June 28-30, 2016, Revised Selected Papers 12*. Springer, 64–74.

[34] Per-Gunnar Martinsson, Gregorio Quintana Ortí, Nathan Heavner, and Robert Van De Geijn. 2017. Householder QR factorization with randomization for column pivoting (HQRRP). *SIAM Journal on Scientific*

Computing 39, 2 (2017), C96–C115.

[35] NVIDIA Corporation. 2024. cuSOLVER. https://docs.nvidia.com/cuda/cusolver/index.html/ Accessed: 2024-11-25.

[36] NVIDIA Corporation. 2024. NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensor-cores/ Accessed: 2024-11-25.

[37] NVIDIA Corporation. 2024. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/ Accessed: 2024-11-25.

[38] Beresford N Parlett and David S Scott. 1979. The Lanczos algorithm with selective orthogonalization. *Mathematics of computation* 33, 145 (1979), 217–238.

[39] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE micro* 17, 2 (1997), 34–44.

[40] Melven Röhrig-Zöllner, Jonas Thies, Moritz Kreutzer, Andreas Alvermann, Andreas Pieper, Achim Basermann, Georg Hager, Gerhard Wellein, and Holger Fehske. 2015. Increasing the performance of the Jacobi–Davidson method by blocking. *SIAM Journal on Scientific Computing* 37, 6 (2015), C697–C722.

[41] Robert Schreiber and Charles Van Loan. 1989. A storage-efficient WY representation for products of Householder transformations. *SIAM J. Sci. Statist. Comput.* 10, 1 (1989), 53–57.

[42] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36, 5-6 (June 2010), 232–240. doi:10.1016/j.parco.2009.12.005

[43] Yaohung M Tsai, Piotr Luszczek, and Jack Dongarra. 2022. Mixed-precision algorithm for finding selected eigenvalues and eigenvectors of symmetric and Hermitian matrices. In *2022 IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Heterogeneous Systems (ScalAH)*. IEEE, 43–50.

[44] University of Tennessee (US). 2024. Matrix Algebra on GPU and Multicore Architectures. https://icl.utk.edu/magma/ Accessed: 2024-11-25.

[45] University of Tennessee (US), University of Manchester (UK). 2024. Parallel Linear Algebra Software for Multicore Architectures. https://github.com/icl-utk-edu/plasma/ Accessed: 2024-11-25.

[46] Lin-Wang Wang, Byounghak Lee, Hongzhang Shan, Zhengji Zhao, Juan Meza, Erich Strohmaier, and David H Bailey. 2008. Linearly scaling 3D fragment method for large-scale electronic structure calculations. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 1–10.

[47] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[48] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. 2011. *QUARK Users' Guide: Queuing And Runtime for Kernels, Version 1.0.* technical report UT-ICL-11-02. University of Tennessee Innovative Computing Laboratory, Knoxville, Tennessee 37996.

[49] Victor Wen-zhe Yu, Jonathan Moussa, Pavel Kůs, Andreas Marek, Peter Messmer, Mina Yoon, Hermann Lederer, and Volker Blum. 2021. GPU-Acceleration of the ELPA2 Distributed Eigensolver for Dense Symmetric and Hermitian Eigenproblems. 262 (2021), 107808. doi:10.1016/j.cpc.2020.107808 arXiv:2002.10991 [physics]