

Pipelined Dense Symmetric Eigenvalue Decomposition on Multi-GPU Architectures

Hansheng Wang

University of Electronic Science and
Technology of China
wanghansheng@std.uestc.edu.cn

Ruiyi Zhan

University of Electronic Science and
Technology of China
rui1@std.uestc.edu.cn

Dajun Huang

University of Electronic Science and
Technology of China
djhuang@std.uestc.edu.cn

Xingchen Liu

University of Chinese Academy of
Sciences
liuxingchen232@mails.ucas.ac.cn

Qiao Li

Xiamen University
qiaoli045@gmail.com

Hancong Duan

University of Electronic Science and
Technology of China
duanhancong@uestc.edu.cn

Dingwen Tao

Institute of Computing Technology,
Chinese Academy of Sciences
taodingwen@ict.ac.cn

Guangming Tan

Institute of Computing Technology,
Chinese Academy of Sciences
tgm@ict.ac.cn

Shaoshuai Zhang

University of Electronic Science and
Technology of China
szhang94@uestc.edu.cn

Abstract

Large symmetric eigenvalue problems are commonly observed in many disciplines such as Chemistry and Physics, and several libraries including cuSOLVERMp, MAGMA and ELPA support computing large eigenvalue decomposition on multi-GPU or multi-CPU-GPU hybrid architectures. However, these libraries do not provide satisfied performance that all of the libraries only utilize around 1.5% of the peak multi-GPU performance. In this paper, we propose a pipelined two-stage eigenvalue decomposition algorithm instead of conventional subsequent algorithm with substantial optimizations. On an 8×A100 platform, our implementation surpasses state-of-the-art cuSOLVERMp and MAGMA baselines, delivering mean speedups of 5.74× and 6.59×, with better strong and weak scalability.

1 Introduction

Eigenvalue decomposition (EVD) aims to factorizes a matrix to a diagonal form: $A = Q \times \Lambda \times Q^T$, $A \in \mathbb{R}^{n \times n}$, where Λ is a diagonal matrix contains the eigenvalues on its diagonal, and Q is an orthogonal matrix consists of eigenvectors. Given a dense symmetric matrix, the entire EVD process typically include three subsequent processes: tridiagonalization [7], iterative solver [18, 22] and back transformation [13]. Among the three processes, tridiagonalization reduces the full matrix to a tridiagonal form; the iterative solver such as QR algorithm [22] and Divide and Conquer (D&C) [18] further diagonalize the tridiagonal matrix and generates the eigenvalues. If the eigenvectors are needed, then the iterative solver also generates the eigenvectors of the tridiagonal matrix, and the back transformation process will form the final eigenvectors.

In dense symmetric EVD, two principal algorithmic families are widely used: one-stage EVD [7] and two-stage EVD [13,

15, 16]. The one-stage approach directly applies Householder reflectors to reduce the original matrix to tridiagonal form. However, this approach relies heavily on BLAS2 operations, which exhibit low arithmetic intensity and suboptimal data reuse. Consequently, it fails to exploit modern GPUs' computing capacity.

To mitigate these inefficiencies, the two-stage approach decomposes tridiagonalization into two phases: successive band reduction (SBR) first reduces the dense matrix to a banded form, and bulge chasing (BC) then converts the band matrix to tridiagonal matrix. By recasting the bulk of the computation into BLAS3 operations, two-stage tridiagonalization achieves up to 10x speedup over one-stage EVD [21]. Thus, this paper focuses on optimizing the two-stage EVD.

In real-world applications, including tight-binding models in condensed matter physics [14], quantum chemistry [4], and density functional theory problems [3], scientists often need to solve large EVD problems that exceed the memory capacity and computational efficiency limits of a single GPU. Consequently, multi-GPU EVD solvers have been developed. Among these, cuSOLVERMp¹ employs a one-stage EVD approach, while libraries such as ELPA [17], MAGMA [19], and SLATE [8] support both one-stage and two-stage EVD methods.

However, as shown in Table 1, given a 49152×49152 matrix, even the state-of-the-art (SOTA) multi-GPU EVD libraries such as MAGMA and cuSOLVERMp achieve only 2.18 TFLOPS and 2.37 TFLOPS on 8 A100 GPUs, respectively. These amounts only reach 1.3% and 1.5% of the GPUs' peak performance (8×19.5 peak TFLOPS), highlighting a significant performance gap in current solutions.

In this paper, we suggest the performance bottlenecks of existing multi-GPU two-stage EVD implementations mainly

¹<https://docs.nvidia.com/cuda/cusolvermp/>

lie in the underutilization of GPU resources, and we thereby propose a pipelined EVD algorithm. The pipeline introduces parallelism across consecutive stages, reduces synchronization overhead and improves GPU utilization. Compared to the SOTA implementations in cuSOLVERMp and MAGMA, the proposed EVD algorithm achieves significant performance improvements and better scalability, demonstrating superior efficiency in large-scale eigenvalue computations on modern multi-GPU architectures.

We consider the contributions of this paper to be:

- We introduce a new pipeline scheme for EVD solvers to fully utilize the GPU resources.
- We design a blockwise data allocation strategy instead of block-cyclic strategy to satisfy the requirement of pipeline, and propose a new load balance approach to provide overall load balance in the pipeline.
- We further optimize the tridiagonalization and back transformation process to reduce the communication volume and improve the kernel performance on GPU architectures.
- We conduct sufficient experiments on the proposed EVD algorithm to show that it has better performance and scalability than the SOTA multi-GPU implementations.

The remainder of the paper is organized as follows: Section 2 introduces the fundamental concepts of EVD. Section 3 demonstrates the motivation, difficulties and solutions of designing pipelined EVD. Section 4 details our implementation and optimizations. Section 5 presents experimental results, while Section 6 discusses related work. Finally, Section 7 concludes the paper and outlines future work.

n	Time		FLOPS	
	MAGMA	cuSOLVERMp	MAGMA	cuSOLVERMp
49152	217.53s	200.02s	2.18T	2.37T

Table 1. Performance comparison of MAGMA and cuSOLVERMp on $8 \times A100$ GPUs for matrix size $n = 49152$, with TFLOPS computed as $\frac{4n^3}{\text{Time}}$.

2 Background

2.1 Two-Stage Eigenvalue Decomposition

2.1.1 Full to Diagonal Steps

There're three consecutive steps to convert a full matrix to a diagonal matrix in two-stage EVD: successive band reduction (SBR), bulge chasing (BC) and iterative solver.

Figure 1 demonstrates the initial two stages of the SBR process, which transforms the original matrix into a band matrix. The algorithm begins by selecting a panel (light yellow block) for QR factorization, resulting in the decomposition $QR(\text{Panel}) = (I - WY^T)R$. This factorization eliminates

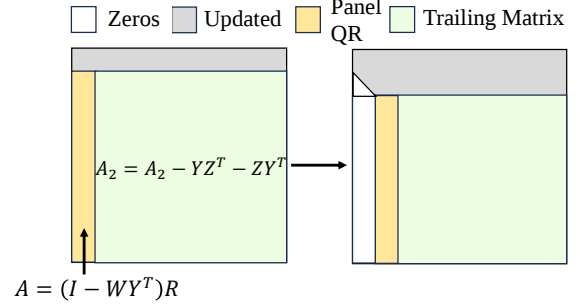


Figure 1. The first two iterations in SBR [21].

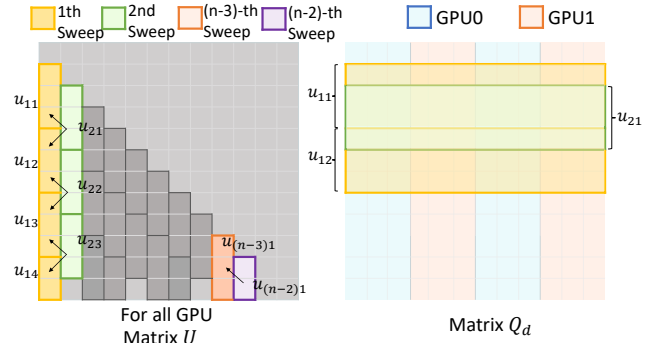


Figure 2. The illustration of BC-Back transformation multiplies Q_d .

off-band elements, with the R matrix replacing the upper triangular portion of the panel. The subsequent trailing matrix update employs a two-sided operation using the ZY representation, which efficiently utilizes the `syr2k` routine [7] as shown in Equation 1.

$$\begin{aligned} Z &= AW - \frac{1}{2}YW^TAW \\ A_2 &= A_2 - YZ^T - ZY^T \end{aligned} \quad (1)$$

Following this update, the modified trailing matrix (now a new full matrix) becomes the input for the next iteration of the SBR process, enabling a recursive reduction of the original matrix to band form.

After the SBR process, BC will be executed to convert the band matrix to a triangular matrix using the similar Householder transformation steps in SBR, but its time complexity is much lower ($O(n^3)$ versus $O(n^2b)$, where b is bandwidth of the band form matrix), and its rich of BLAS2 operations, more details can be found in [21].

When BC is finished, the tridiagonal matrix T will be obtained, the iterative solvers such as D&C [18] and QR algorithm [22] will be performed to diagonalize the matrix T to Λ with the eigenvalues presented on its diagonal.

2.1.2 Back Transformation

When only the eigenvalues are desired, the EVD stops at the end of the iterative solver; while the eigenvectors are also required, back transformation will be taken. For two-stage EVD, back transformation comprises SBR-Back and BC-Back. While SBR-Back follows the one-stage EVD approach, BC-Back faces strict data dependencies requiring updates in a specific order (u_{ij} after $u_{(i+1)(j-1)}$ and $u_{(i+1)j}$), with a complexity of approximately $2n^3$ FLOPS (see Figure 2 for detail). The standard BC-Back implementation lacks BLAS3 operations, limiting performance. MAGMA [15] optimizes this by grouping Householder vectors via LAPACK’s `larft` routine and performing GEMM-based updates (Figure 3), but BC-Back remains the primary bottleneck, reducing the performance advantage of two-stage EVD over one-stage EVD from $6.1\times$ to $1.2\times$ [21].

2.2 EVD Solvers on Multi-GPU Architectures

Multiple linear algebra libraries support computing large scale EVD problems, including ScaLAPACK [2], Eigen [11], Intel MKL on multi-CPU, and cuSOLVERmp, MAGMA [19], SLATE [8] and ELPA on multi-GPU or hybrid architectures. Basically, on multi-GPU architectures, the matrix are distributed on different GPUs with a 1D block-cyclic style or a tiling style to improve the load balance. Specifically, in terms of the two-stage EVD implemented in MAGMA, SLATE and ELPA, the EVD solvers compute SBR, BC, D&C, BC-Back and SBR-back in sequence.

However, based on our experiments, all of the EVD solvers on multi-GPU architectures cannot provide good performance and scalability. For example, MAGMA only reaches less than 2% TFLOPs of the theoretical peak multi-GPUs performance, and using 2 or 4 GPUs even leads to worse performance than single GPU due to the expensive communication overhead. Similar phenomenon is also observed in cuSOLVERmp and SLATE. ELPA is an exception that it keeps good weak and strong scalability, but its performance on 4 GPUs is lower than MAGMA or cuSOLVERmp on one GPU.

3 Motivation, Difficulties and Solutions in Pipeline

3.1 Motivation

Figure 4 presents the breakdown of the two-stage EVD on 4 A100 GPUs. MAGMA treats the stages of EVD as independent tasks: for example, BC starts after SBR, and BC-Back is executed at the end of D&C. This design leads to two issues. First, when performing BC and BC-Back-Pre (forming T matrices with the LAPACK `larft` routine), the GPUs are idle. Second, SBR suffers from load imbalance, where some GPUs wait for others to finish their SBR tasks. Migrating BC and BC-Back-Pre from the CPU to the GPUs using recent techniques [21] can mitigate the first issue, however it introduces

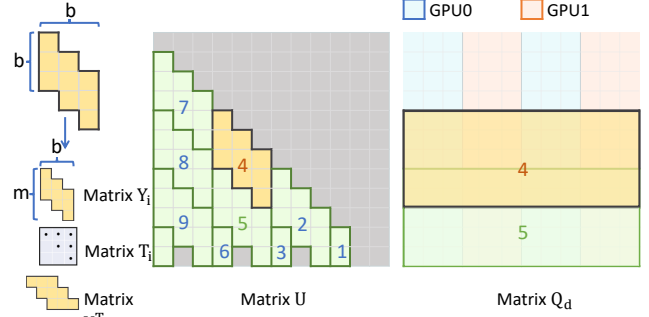


Figure 3. The illustration of BC-Back transformation multiplies Q_d .

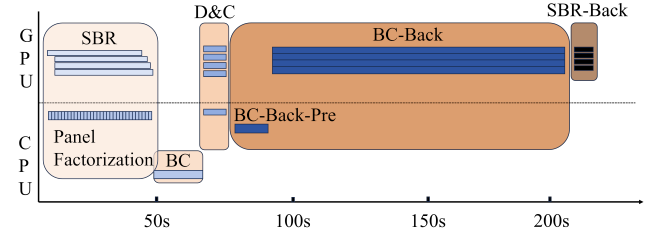


Figure 4. The timeline of MAGMA two-stage EVD routine with matrix size 49152×49152 on 4 A100 GPUs.

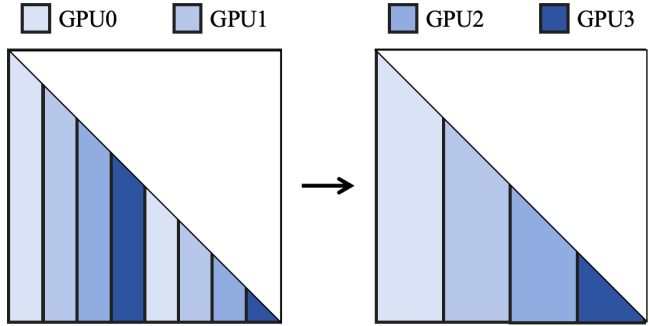


Figure 5. The difference between the block cyclic distribution and the blockwise distribution.

severe load imbalance (see Section 4.2). Observing opportunities to overlap computations across stages, pipelining is effective for two reasons: 1) it has the potential to maintain high GPU utilization, and 2) it enables load balancing across the entire EVD pipeline rather than within a single stage, providing more flexibility to adjust per-GPU load. Converting the conventional sequential EVD into a pipelined EVD, however, poses several challenges, which we address in the following sections.

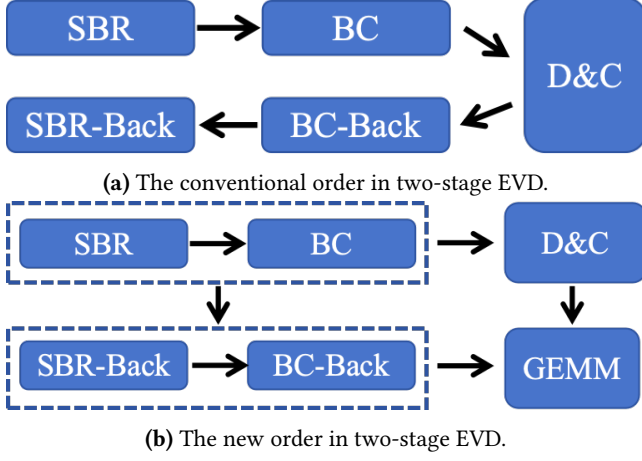


Figure 6. The comparison between conventional order and the proposed order in two-stage EVD.

3.2 Blockwise Distribution

Directly converting conventional EVD solvers into a pipelined EVD solver is not easy, because conventional EVD solvers typically adopt a block-cyclic scheme; for example, MAGMA uses a 1D block-cyclic scheme, and cuSOLVERMp employs a 2D block-cyclic scheme. We use SBR as an example to illustrate the difficulty. Figure 5 provides an illustration of MAGMA’s implementation. We observe that GPU0 holds different blocks of A , and it will always be busy until the final block on GPU0 is computed. However, the BC process can start as soon as the first few blocks have been updated and computed, and recent fast GPU-based BC implementations require the full GPU resources, especially memory bandwidth [21]. Although BC can start early, there are still bottlenecks in the block-cyclic scheme: 1) BC cannot fully utilize GPU resources; 2) expensive communication is required to move the band-form matrix between different GPUs; and 3) it is hard to schedule tasks across multiple stages.

Therefore, we abandon the conventional block-cyclic distribution and distribute the matrix using a column-blockwise scheme, as shown on the right of Figure 5. Compared with the block-cyclic distribution, the blockwise distribution allows us to create a pipeline. For instance, GPU0–GPU3 are busy until the first block in SBR finishes, because all of the GPUs participate in the computations of forming the matrix Z and updating the trailing matrix, which are the most expensive parts of SBR. Once the first block on GPU0 is done, GPU0 can start the following BC stage immediately. With this matrix distribution strategy and the corresponding pipeline, overlap between different stages becomes possible, leading to better utilization of GPU resources.

3.3 Reordered Stages

Another difficulty of the pipelined EVD solver is that the D&C process is hard to be assembled into the pipeline. According to Figure 3, we can find that BC-Back of the first block requires the last several rows in the orthogonal matrix Q_d generated by the D&C process. Unfortunately, Q_d cannot be obtained before the finalization of D&C, because Q_d continues to be updated until the last conquer process, resulting in the stop of the pipeline. To avoid such dependency, we consider to schedule the order of the stages in a different way, based on a simple observation that the D&C process is actually an independent process to other stages.

Assuming BC is completed and the tridiagonal matrix is obtained, in the conventional order (Figure 6a), D&C will be computed at first to generate the Q_d matrix, and BC-back applies Householder vectors to Q_d to Q_{bd} , and SBR-back further applies the WY representation on Q_{bd} to get the final orthogonal matrix Q . Indeed, if we regard the D&C as an independent stage that it can start as soon as the tridiagonal matrix is obtained, then we can bypass the D&C process and directly move to the SBR-Back and BC-back process to maintain the pipeline, as illustrated in Figure 6b. However, as Q_d is not available during the back transformation process, the Householder vectors will be applied on an identity matrix instead of Q_d , resulting in an extra GEMM to form the final eigenvectors. Fortunately, the GEMM is square and large, which is efficient on modern GPU architectures, so that the extra overhead is tolerable. Furthermore, as the CPU is almost always idle, we can let the CPU handle D&C, while the back transformation is executed on GPUs, so that the D&C can be overlapped ideally.

3.3.1 Correctness of Reordered Stages

One of the problems of reordered stages is that it is unknown whether this modification can still provide correct result. Thus, we try to prove that both the orthogonality and backward error are still bounded by $O(\epsilon)$, where ϵ is the rounding-off error. For orthogonality, as the Householder operations guarantee $Q_{sb} = Q_s Q_b$ ’s orthogonality and Q_d preserves orthogonality via its traditional D&C computation, we only need to prove the orthogonality of the GEMM operation $Q = Q_{sb} Q_d$. Let $E = \text{fl}(Q_1 Q_2) - Q_1 Q_2$. Then:

$$\text{fl}(Q_1 Q_2) \text{fl}(Q_1 Q_2)^T - I = (Q + E)(Q + E)^T - I = EQ^T + QE^T + EE^T.$$

Therefore:

$$\|\text{fl}(Q_1 Q_2) \text{fl}(Q_1 Q_2)^T - I\|_2 = \|EQ^T + QE^T + EE^T\|_2 \leq 2\|EQ^T\|_2 + \|EE^T\|_2 \leq 2\|E\|_2 + \|E\|_2^2.$$

From [10], we have:

$$\|\text{fl}(QA) - QA\|_F \leq \sqrt{n}\epsilon \|A\|_F.$$

When A is orthogonal, $\|A\|_F = \sqrt{n}$, and $\|A\|_2 \leq \|A\|_F$, so:

$$\|E\|_2 \leq \|E\|_F \leq n\epsilon.$$

Since orthogonal transformations preserve the 2-norm, we have $\|EQ^T\|_2 = \|E\|_2$. Therefore:

$$\|\text{fl}(Q_1 Q_2) \text{fl}(Q_1 Q_2)^T - I\|_2 \leq 2n\epsilon + n^2\epsilon^2.$$

The relative orthogonality error:

$$\frac{\|\text{fl}(Q_1 Q_2) \text{fl}(Q_1 Q_2)^T - I\|_2}{n} \leq 2\epsilon.$$

This shows the GEMM $Q_1 Q_2$ remains orthogonality. For backward stability:

$$\text{fl}(Q_1 Q_2) \Sigma \text{fl}(Q_1 Q_2)^T - Q \Sigma Q^T = E \Sigma E^T + E \Sigma Q^T + Q \Sigma E^T.$$

where $Q = Q_1 Q_2$, $E = \text{fl}(Q_1 Q_2) - Q$. Then:

$$\|\text{fl}(Q_1 Q_2) \Sigma \text{fl}(Q_1 Q_2)^T - Q \Sigma Q^T\|_2 \leq \|E \Sigma E^T\|_2 + 2\|\Sigma E\|_2.$$

Since Σ is diagonal, $\|E\|_2 \leq n\epsilon$, we derive:

$$\|\text{fl}(Q_1 Q_2) \Sigma \text{fl}(Q_1 Q_2)^T - Q \Sigma Q^T\|_2 \leq (2n\epsilon + n^2\epsilon^2)\|\Sigma\|_2.$$

Therefore:

$$\frac{\|\text{fl}(Q_1 Q_2) \Sigma \text{fl}(Q_1 Q_2)^T - Q \Sigma Q^T\|_2}{n\|Q \Sigma Q^T\|_2} \leq 2\epsilon + n\epsilon^2 \approx 2\epsilon,$$

indicating the backward stability is preserved.

Experimentally, Table 2 compares the backward error $\frac{\|A - Q \Lambda Q^T\|}{n\|A\|}$ and orthogonality $\frac{\|I - Q Q^T\|}{n}$ between reordered stages EVD and cuSOLVER Dsyevd routine with matrix size 16384×16384 . And the results reveal that conventional EVD has slightly better stability and orthogonality than reordered stages EVD, but reordered stage EVD still preserves the FP64 accuracy, which conforms the above mathematical proof.

Distribution	Backward		Orthogonality	
	cuSOLVER	Ours	cuSOLVER	Ours
Cluster0	5.5E-19	1.4E-18	8.2E-17	1.5E-16
Cluster1	4.4E-19	3.9E-19	7.9E-17	1.4E-16
Geometric	2.5E-19	8.6E-19	6.7E-17	1.4E-16
Arithmetic	5.6E-19	1.3E-18	8.5E-17	1.5E-16
Normal	3.8E-19	6.7E-19	4.0E-17	1.5E-16
Uniform	4.9E-19	1.8E-18	4.0E-17	1.5E-16

Table 2. The EVD accuracy comparison between cuSOLVER and the proposed reordered stages EVD with different matrix types. For Cluster0, Cluster1, Geometric and Arithmetic distribution, condition number is $1E8$ and the largest eigenvalue is $1E6$.

3.4 Load Balance and Overall Pipeline

However, there is also an issue existing in the proposed method, that it has severe load imbalance. This is because GPU0 only needs handle the Block0, while GPU1 not only handles the trailing matrix update in Block0, but also needs to handle its own computations. Additionally, the BC process also has imbalance load. Thus, regarding tridiagonalization, GPU_i always undertakes more computations than GPU_{i-1} .

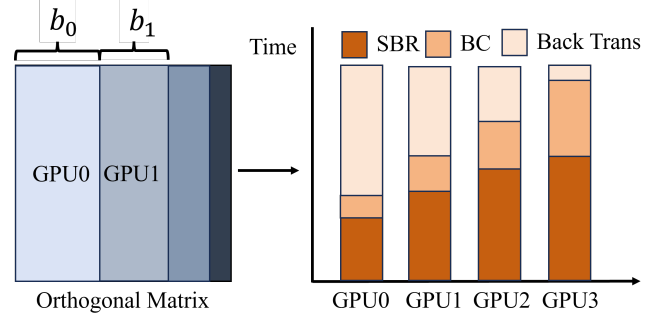


Figure 7. The illustration of adjustable blocksize in SBR-Back and BC-Back.

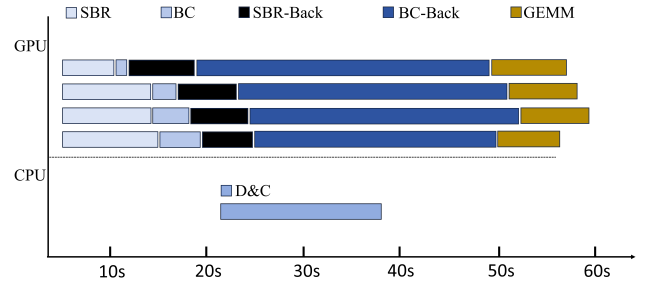


Figure 8. The overall pipeline of the proposed EVD solver.

Fortunately, benefiting from the reordered-stage strategy which detaches the D&C process, we have an opportunity to leverage SBR-Back and BC-back to balance the GPU load, and the idea shown in Figure 7 is natural and simple. Compared to SBR and BC, the back transformation process including SBR-Back and BC-Back has less dependency, which means we have more flexibility to form the eigenvectors. As Figure 7 shows, suppose there are 4 GPUs, we can dynamically choose $b_0 > b_1 > b_2 > b_3$, so that the GPU_i performs more computations than other GPUs than GPU_{i+1} . Hence, demonstrated by the bar figure in Figure 7, although GPU0 spends less time on SBR and BC, we can let GPU0 do more computations in back transformation, leading to end-to-end load balance. In practice, we keep the per-GPU adjustment within 5% of a base block size to provide enough slack to equalize the timeline.

After adjusting the blocksize in back transformation process, the rough overall pipeline is shown in Figure 8. Compared to MAGMA, we do not pursue the load balance in a single stage, instead, we try to leverage the back transformation to balance the load with tridiagonalization. As a result, although the tridiagonalization process has severe load imbalance, the overall load can be balanced.

Finally, the initial pipelined two-stage EVD algorithm (Algorithm 1) employs MPI with one process per GPU (identified by *rankID*). Each GPU retrieves its local A submatrix (Lines 5–6). The root process (rank 0) executes SBR immediately

(Lines 8–14), while others wait for preceding ranks. Crucially, after the completion of SBR, the GPU immediately proceeds to BC without waiting for the final finish of SBR, and then concurrently accumulating W for SBR-Back (Lines 15–16). Since BC (it is inherently fast) always completes before SBR-Back, BC-Back can initiate immediately. Leveraging CPU idle time during SBR-Back and BC-Back, rank 0 spawns a D&C thread (Lines 29–32) for parallel execution. And after D&C and BC-Back’s finalization, the final GEMM will assemble the global eigenvector matrix Q .

4 Implementation and Optimization

Although the pipelined design of EVD provides a better utilization of GPU resources, the implementations and optimizations are also important to further improve the performance. Therefore, in this section, we’ll discuss our implementation and optimization details on some stages.

4.1 Communication Avoiding SBR

Compared to SBR on a single GPU, SBR on multi-GPUs is more challenging due to communication on the trailing matrix. Considering the computations of forming $Z = AW - \frac{1}{2}YW^TAW$ and based on our previous design, A is distributed on GPUs blockwisely, and W is already broadcast to all GPUs. Then computing AW is challenging if only the lower triangular part of A is stored, because expensive communication always exist if GPU_{*i*} doesn’t hold the corresponding rows in the upper triangular part of A . Figure 9 gives an explanation, suppose GPU2 undertakes a sub-task, to deliver the correct results, GPU2 needs the data from GPU0 and GPU1 to fill in the upper part of the local matrix, leading to communication. Furthermore, this kind of communication happen very frequently because AW will be recomputed after every panel factorization. Quantitatively, if the bandwidth is b and the matrix size is $n \times n$, then the total communication words will be $\sum_{i=1}^{\frac{n}{b}-1} \frac{1}{2}(n-i*b)^2 = \frac{n(n-b)(2n-b)}{12b}$, which is intolerable.

To solve this problem, one can consider to distribute the entire matrix rather than a triangular matrix, so that the local matrix on each GPU can hold the entire matrix. The disadvantage is that the trailing matrix updates using syr2k will be substituted with GEMM, thereby the FLOPS are increasing to 2x, but we demonstrate the scarification is worthy. We can specify GPU_{*i*} as an example. If the matrix is stored as a full matrix, then the extra computation cost of trailing matrix update will be $t_1 = \frac{2(ik-xb)kb}{p}$, where k is the number of columns each GPU takes, b is the bandwidth in BC, x is the round of panel factorization in SBR and p is the TFLOPs of the GEMMs. Then the communication time will be $t_2 = \frac{8(ik-xb)*k}{q}$, where q is the network bandwidth. If $t_1 < t_2 \rightarrow b < \frac{4p}{q}$, then the extra time cost of computations will be less than the communication using triangular form. On the modern GPU architectures, computing AW typically can reach 10 TFLOPs, while $q \approx 0.35 \text{ TB/s}$ on H100-SXM

Algorithm 1 The Pipelined Multi-GPU EVD

Input: Matrix A – the original dense symmetric matrix.
 Int n – the row or column numbers of matrix A .
 Int $ngpu$ – the number of GPUs.

Output: Array Λ – the eigenvalues of matrix A .
 Matrix Q – the orthogonal eigenvector matrix of A .

```

1: % Initialize the MPI and get the process rank.
2: rankID ← MPI_Init()
3: n_cols ←  $\frac{n}{ngpu}$ 
4: % Partition the data using block-wise style and transfer each block to GPU.
5: Aih = A(:, rankID * n_cols : (rankID + 1) * n_cols)
6: Aid = cudaMemCopy(Aih)
7: % SBR subprocess
8: if 0 == rankID
9:   SBR(Aid)
10: else
11:   % wait the finish of rankID – 1’s SBR.
12:   waitSBRFinishedEvent(rankID – 1)
13:   SBR(Aid)
14: endif
15: T, Ui = BC(Aid) % BC subprocess
16: Wi = SBR_Back_genW(Wid) % SBR_Back_genW subprocess
17: % BC subprocess
18: for k = 1 : 1 : ngpu do
19:   QisT = SBR_Back_genQ()
20:   waitSBR_Back_genWFinishedEvent(k + 1)
21: end for
22: % wait for the finish of the last GPU’s BC.
23: waitBCFinishedEvent(ngpus)
24: % Collect the U matrices generated by BC on each GPU.
25: U = MPI_gather(Ui)
26: % BC-Back subprocess
27: QibsT = BC(U)
28: % The rankID 0 spawns a thread for D&C, enabling concurrent execution with back transformation.
29: if 0 == rankID
30:   Z, Λ = std :: threadDC_thread(CPUDC(T)
31:   std :: thread.join())
32: endif
33: % Synchronize all Ranks.
34: MPI_barrier();
35: % Final GEMM
36: for k = 1 : 1 : ngpu do
37:   Zi = cudaMemCopyAsync(Z(:, (k – 1) * n_cols : k * n_cols))
38:   Qi(k * n_cols : (k + 1) * n_cols, :) = cudaGEMM(Zi, Qibs)
39: end for
40: % Assemble the global eigenvector matrix Q.
41: Q(rankID * n_cols : (rankID + 1) * n_cols, :) = Qi
42: MPI_Finalize

```

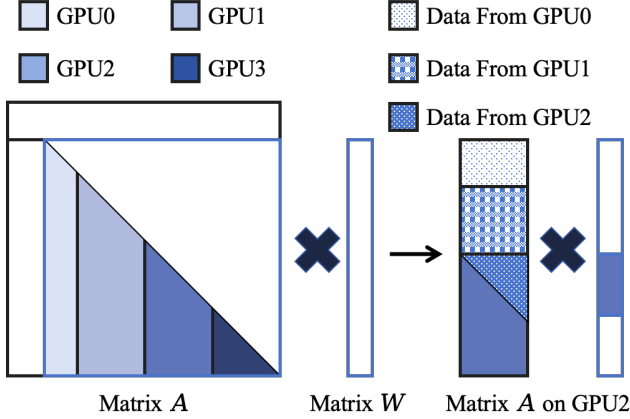


Figure 9. The communication behavior when only storing the lower triangular part of matrix A .

GPU with Gen4 Nvlink one direction. Thus, if $b < 114$, the proposed SBR spends less time. Typically, in two-stage EVD, b is usually set to 32 [21] to deliver best performance, which indicates performing extra computations in SBR is worthy, because the inter-GPU communication are more expensive. Note that using GPUs without Nvlink leads to smaller q , thereby amplifying the advantage of the proposed SBR.

After this optimization, the trailing matrix A doesn't need any communication, while only the matrices W , Y and Z are expected to broadcast, which accounts for $\sum_{i=1}^{\frac{n}{b}-1} 3(n-i*b)*b$. To further improve the performance, the GEMM is upgraded to syr2k on the last GPU, as the last GPU doesn't need data from other GPUs anymore. Also, we adopt double blocking SBR [21] to accelerate the trailing matrix update process.

4.2 Distributed BC

Compared to distributed SBR, BC is a more difficult task to be distributed. As a result, even the recent BC implementations on distributed computer architectures including ELPA [17] and MAGMA [19] still handle the BC problem on a single CPU, which is slow and a waste of hardware resource. Wang et.al. [21] proposes a fast single GPU-based implementation, but we suppose that directly employing the same implementation (each warp handles one BC sweep) incurs large communication overhead and synchronization problems. For example, if we let each GPU handle a group of sweeps, then every GPU will have to load the entire band matrix B . Besides, as each Householder transformation in BC will affect the next $2b \times b$ block in B , then all of the GPUs have to update this block, thereby resulting in large communication overhead.

Thus, we propose a new BC algorithm on distributed GPU architectures without large communication, and the idea is shown in Figure 10. To simplify the illustration, we use two GPUs as an example, and suppose GPU0 and GPU1 takes Block0 and Block1 from band form matrix respectively. GPU0

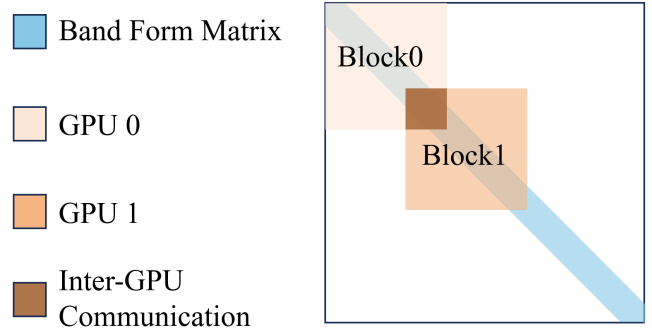


Figure 10. Illustration of distributed BC implementation.

starts performing BC after the finalization of SBR task on GPU0, while GPU1 is still working on its own SBR task meanwhile. Once BC on GPU0 reaches the bottom right of Block0 (brown block in Figure 10), GPU1 can finish its own SBR task and wait for synchronization. As Figure 10 shows, the brown block is an overlap of Block0 and Block1, thereby needing inter-GPU communication to deliver the up-to-date data to GPU1. Fortunately, it is the only communication between subsequent GPUs, while other distribution strategies such as block-cyclic require communication between all GPUs on the entire band matrix.

4.3 BLAS2-based BC-Back

The aforementioned BC-Back implemented in MAGMA converts BLAS2 to BLAS3 operations to improve the performance. However, as Figure 3 shows, MAGMA's BC-Back relies on LAPACK larft routine to form the YTY^T , and we can find that T is a triangular matrix, and Y is padded with zeros. As a result, extra computations are taken, and the actual FLOPS increase from $2mn^2$ to $4mn^2$. Although BLAS3 has better performance than BLAS2, it is unknown if increasing the FLOPS is worthy. Based on our testing on GEMMs in MAGMA's BC-Back, we find that, due to the special GEMM shapes (e.g. two dimensions are 64 and another dimension is very large), the GEMMs can only reach up to 6 TFLOPs on H100 GPU, while the peak performance is 67 TFLOPs. Besides, another overhead is typically overlooked that calling LAPACK larft is not cheap and it typically costs 20% time of BC-Back. In this case, we have an opportunity to design an elaborate BC-Back kernel using BLAS2 operations, it will be potentially faster than MAGMA if the BLAS2 operations can reach 3 TFLOPs.

Figure 11 details the design of BLAS2-based BC-Back. The matrix U stores the Householder vectors generated from BC, and the matrix Q_s^T represents the orthogonal matrix from SBR-Back. As BC-Back has strict data dependency within U matrix, we follow the bottom to top and left to right order to update Q_s^T . To further improve the performance, on a single GPU, we optimize the CUDA kernel carefully using the following techniques.

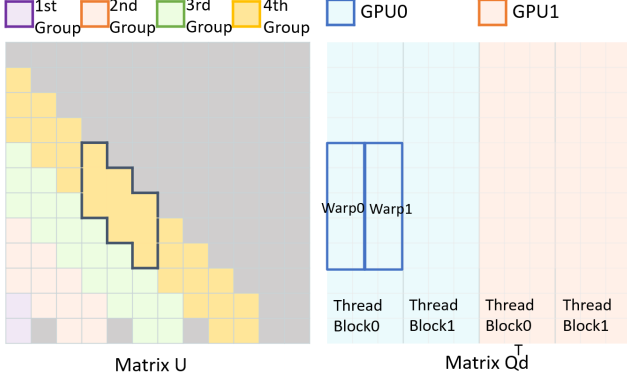


Figure 11. The proposed BC-Back implementation.

- Collecting several Householder vectors into a group and loading several groups into shared memory together. For example, as shown in Figure 11, we load G_1B_1 , G_2B_1 , G_3B_1 and G_4B_1 into shared memory at the same time, so that they can be reused until all columns in Q_s^T are updated.
- Loading Q_s^T into registers instead of shared memory. Compared to matrix U , Q_s^T is updated frequently and needs to be exchanged between register files and global memory one round after another. Therefore, storing Q_s^T in register files has two distinct advantages: 1) save more shared memory space for storing U , and 2) moving data from global memory to registers is faster than that to shared memory on many GPU architectures.
- Avoiding bank conflicts in shared memory. The length of Householder vectors in U is typically a multiple of 32, so that it is easy to incur bank conflicts. To solve this problem, we rearrange the elements in U .

With the above techniques, our BC-Back kernel provides 1.5x speedup compared to MAGMA’s BC-Back on 1 A100 GPU, as shown in Figure 12. Besides, based on the experiments, we find MAGMA’s BC-Back does not have strong scalability, resulting in 12.3x speedup on 8 A100 GPUs. This is probably because the BLAS3 implementation in MAGMA generates Y and T matrices (as depicted in Figure 3), thereby increasing communication and leading to difficulties on hiding the data movement latency.

5 Evaluation

Experiments were conducted on two Multi-GPU architectures: an 8-GPU NVIDIA A100 PCIe platform and an 8-GPU NVIDIA H100-SXM platform. Both platforms utilize identical 48-core Intel® Xeon® Platinum 8468 processors and ran Debian GNU/Linux 12 (bookworm). All evaluations employed the NVIDIA HPC SDK 25.3, which integrates a C++ compiler with optimized cuBLAS, cuSOLVER and NVSHMEM libraries.

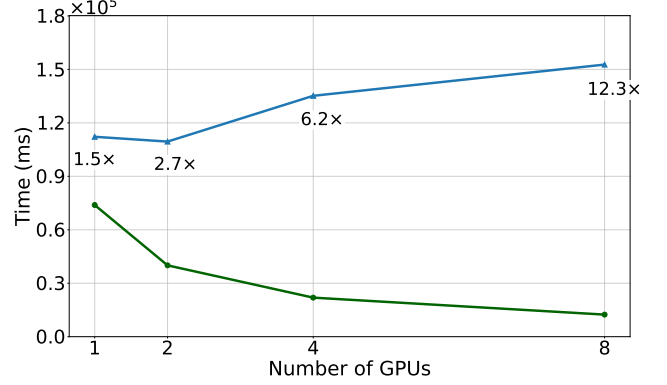
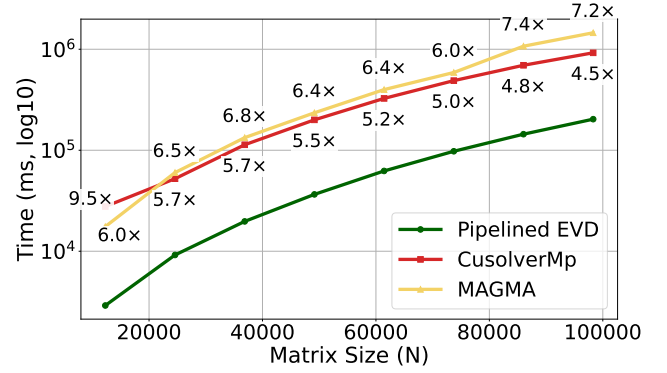
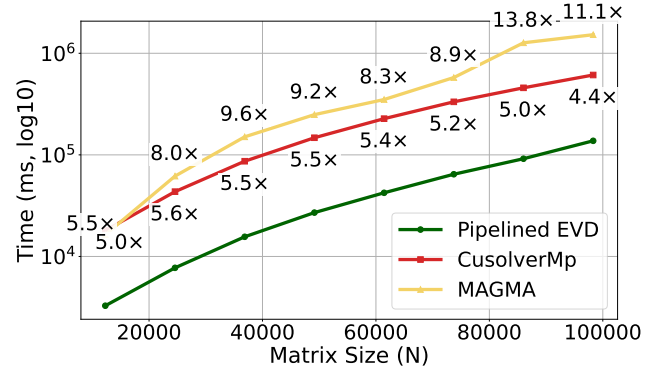


Figure 12. The performance comparison between MAGMA BC-Back and proposed BC-Back on A100 GPUs. Matrix size is 49152×49152 .

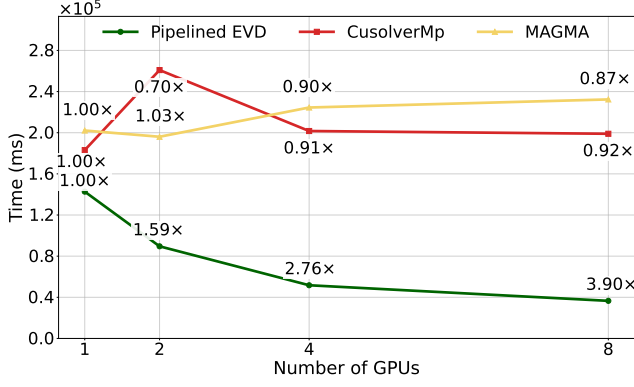


(a) The overall EVD performance on 8 A100 GPU.

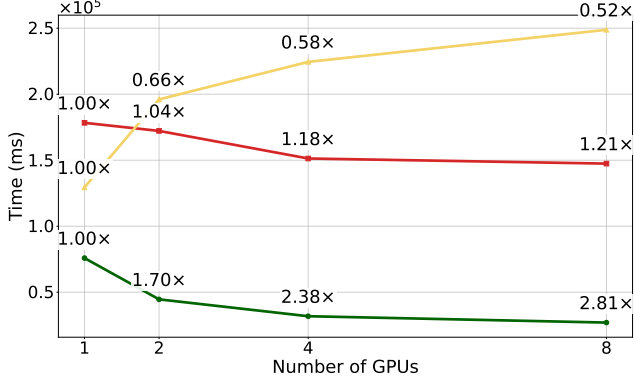


(b) The overall EVD performance on 8 H100 GPU.

Figure 13. The overall performance comparison between cuSOLVERMp, MAGMA, and pipelined EVD for different matrix sizes on 8 A100/H100 GPUs.



(a) The strong scalability comparison on A100 GPUs.



(b) The strong scalability comparison on H100 GPUs.

Figure 14. The strong scalability comparison among cuSOLVERMp, MAGMA, and the our proposed pipelined EVD.

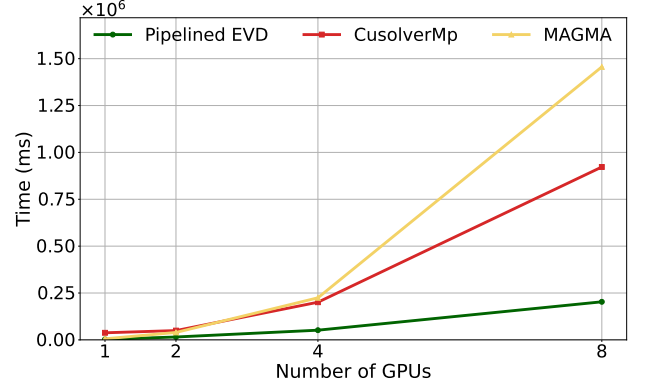
5.1 Overall Performance

We compare the proposed pipelined EVD algorithm against two implementations, cuSOLVERMp and MAGMA in Figure 13. In short, the pipelined EVD demonstrates consistent and substantial performance gains on A100 and H100-SXM GPUs. On the A100 GPUs, it achieves 5.74× and 6.59× speedup over cuSOLVERMp and MAGMA; while on the H100 GPUs, it delivers 5.25× and 9.24× speedup relative to cuSOLVERMp and MAGMA, respectively.

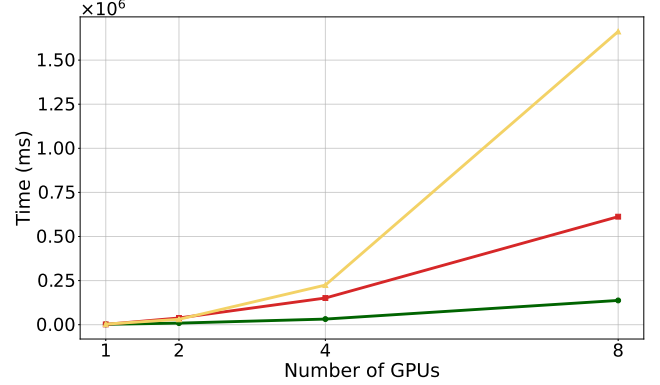
These results validate two critical advantages of our approach: first, the algorithmic efficiency derived from the pipelined design, which significantly reduces synchronization overhead while maximizing GPU utilization through an optimized workflow; and second, the hardware generality demonstrated by consistent performance gains across fundamentally different GPU architectures, confirming that its portability is not limited to any specified GPU architecture.

5.2 Scalability

We evaluate strong scaling performance against cuSOLVERMp and MAGMA on 1, 2, 4, 8 GPU configurations for a 49, 152 ×



(a) The weak scalability comparison on A100 GPUs.



(b) The weak scalability comparison on H100 GPUs.

Figure 15. The weak scalability comparison among cuSOLVERMp, MAGMA, and the our proposed pipelined EVD.

49, 152 symmetric matrix in Figure 14, and the numbers denote the speedups compared to the elapsed time on 1 GPU. As cuSOLVERMp cannot handle such matrix on one GPU, we use cuSOLVER's Dsyevd instead. As shown in Figure 14, both benchmarks exhibit poor strong scalability on A100 GPUs: MAGMA and cuSOLVER fail to scale. On H100, MAGMA demonstrates negative strong scaling, exposing the limitations of hybrid CPU-GPUs implementations. cuSOLVERMp only exhibits marginal scaling, and it is probably because one-stage EVD has limited arithmetic intensity [21, 23], which is hard to overlap data movements. In contrast, our pipelined EVD algorithm maintains robust scalability across both platforms. This consistent performance across hardware generations validates the effectiveness of our pipeline design.

The weak scalability is illustrated in Figure 15. The EVO solvers are executed across 1, 2, 4 and 8 GPUs with problem sizes scaled proportionally: 12288×12288 (1 GPU), 24576×24576 (2 GPUs), 49152×49152 (4 GPUs), and 98304×98304 (8 GPUs). Based on the results, our algorithm demonstrates much better weak Scalability than cuSOLVERMp and MAGMA, near-perfect weak scalability on both A100 and H100 platforms,

confirming that the pipelined EVD delivers superior scalability for large-scale eigenvalue problems.

To sum up, the proposed pipelined EVD demonstrate better scalability than cuSOLVERmp and MAGMA. We think there are two main reasons. First, pipelined EVD has less synchronization across different GPUs in the tridiagonalization process (SBR and BC). In other words, each GPU concentrates on its own task and minimizing inter-GPU synchronization. Second, the largest bottleneck in two-stage EVD is the BC-Back, and the BLAS2-based implementation provides less communication than BLAS3-based implementation, leading to better BC-back scalability (Figure 12).

5.3 Comparison with Block-Cyclic Implementation

To rigorously validate the efficiency of the pipelined EVD, we implement a block-cyclic EVD. Based on previous analysis, the block-cyclic strategy is difficult to leverage pipeline, so that the stages are executed in sequence. Compared to MAGMA’s block-cyclic EVD, the new baseline involves the same optimizations as pipelined EVD uses, including communication avoiding SBR, BLAS2-based BC-Back and reordered stages. (We empirically demonstrate that reordering the stages yields better performance.) Figure 16 shows the timeline of block-cyclic baseline on 4 A100 GPUs with a 49152×49152 matrix.

Compared to Figure 8, the key distinction lies in the SBR and BC process. The SBR of block-cyclic baseline is more balanced, but it is longer than imbalanced pipelined SBR. This can be attributed to two reasons. 1) Block-cyclic SBR yields more communication: it always needs data movements across all GPUs, while pipelined SBR no longer requires communication from GPU_i if GPU_i has finished its own task. 2) Block-cyclic SBR cannot degrade to single GPU SBR on the last GPU, while pipelined SBR on the last GPU direct uses syr2k instead of GEMM. For the BC process, the pipelined EVD leverages back transformation to utilize all of the GPUs, while in block-cyclic BC, GPUs except GPU0 are idle and they’re waiting for the finalization of BC on GPU0. These advantages enable the pipelined EVD to achieve 55.4s execution time versus 63.4s for the block-cyclic implementation, making the motivation of pipelined EVD more reasonable.

6 Related Work

Computing symmetric/Hermitian eigenvalue problems has a long history. In 1980s, LAPACK [1] includes syevd routine using the blocking one-stage tridiagonalization method [7], but it is limited on single CPU platform. Real world applications such as density functional theory problems [3] demand dense large EVD solvers with matrix size over 100k, which cannot be solved by one single CPU. Thus, in 1990s, scaLAPACK [2] and was proposed to solver larger EVD problems on distributed CPU architectures. Further, with the development of multicore architectures, PLASMA [6] and Eigen [11]

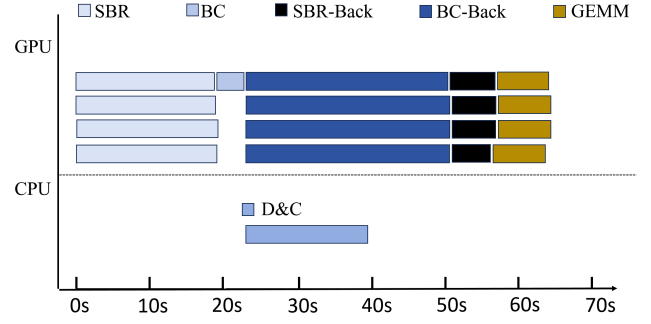


Figure 16. The timeline of block-cyclic EVD on a 49152×49152 matrix using proposed optimizations on 4 A100 GPUs.

emerges to utilize the computing capacity of multicore CPUs. Later in 2010s, the emerging GPUs demonstrate superior performance over CPUs on dense linear algebra problems, and MAGMA [19] builds a bridge between CPU and GPU that it provides EVD solvers on hybrid CPU-GPU architectures. Nvidia also develops their own EVD solvers including cuSOLVER and cuSOLVERmp, which target on single GPU and multi-GPU architectures respectively.

The symmetric/Hermitian EVD solvers contain one-stage and two-stage EVD. one-stage EVD is more suitable for small matrices, because its time complexity is lower than two-stage EVD and most of the matrix data can fit into cache [7]. two-stage EVD [13, 15, 16, 25] converts many BLAS2 operations in one-stage EVD, but it increases the time complexity from $O(4n^3)$ to $O(6n^3)$ because BC-Back contributes $2n^3$ extra computations [9]. Fortunately, the increased BLAS3 operations compensate the growing complexity, and the recent research on two-stage tridiagonalization demonstrate over 10x speedup over cuSOLVER’s one-stage tridiagonalization [21].

The applications of large symmetric/Hermitian EVD solvers mainly lie in Physics and Chemistry, and there are already some scientific computing software assembles EVD solvers, including VASP [12], BerkeleyGW [5] and Gromacs [20]. It is noteworthy that these software typically adopts ELPA [17, 24], which support both one-stage and two-stage on multiple CPU-GPU architectures, as their built-in EVD solvers.

7 Conclusion

In this paper, we propose a pipelined EVD solver on multi-GPU architectures. The motivation of pipelining the EVD solver is that conventional two-stage EVD implementations do not have load balance and lack utilization of GPU resources, which can be solved by pipeline potentially.

However, there are some difficulties on converting the existing two-stage EVD solver to pipelined solver. The first is that the conventional block-cyclic data distribution strategy does not allow as to pipeline SBR and BC, as BC demands the full GPU memory bandwidth and the communication between SBR and BC is intolerable. The second is that D&C

keeps updates the orthogonal matrix until the final conquer process, resulting in a pipeline stall. Therefore, to make the pipeline work, we change the distribution to blockwise distribution and then regard D&C as an independent task, to maintain the pipeline without D&C. Nevertheless, these modifications incur much more severe load imbalance, but we found it can be solved by adjusting the blocksize in SBR-Back and BC-Back, thereby balance the load in SBR and BC. In other words, we consider the four stages to be a whole task in terms of load balance. Experimentally, the proposed pipelined EVD solver utilizes the GPUs more efficiently.

To further improve the performance, we propose several optimization techniques to reduce the communication in distributed SBR and BC. We also found the BLAS3-based BC-Back is inefficient because it increases the time complexity and the sizes and shapes of the GEMMs are too special to be executed efficiently. Thus, we use BLAS2-based BC-Back on multiple GPUs which is over 3.0x faster than conventional BC-Back implementation. Eventually, our implementation surpasses cuSOLVERmp and MAGMA baselines, delivering mean speedups of 5.74x and 6.59x, respectively on A100 GPUs.

Our future work lies on enlarge the scale of EVD problems, as there are also some applications require decomposing $1M \times 1M$ symmetric/Hermitian matrices, which can only be solved on supercomputers with thousands of GPUs. Additionally, migrating the proposed techniques to non-symmetric matrices will be another challenge topic.

References

- [1] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide*. SIAM.
- [2] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petit, et al. 1997. *ScaLAPACK users' guide*. SIAM.
- [3] H Chermette. 1998. Density functional theory: a powerful tool for theoretical studies in coordination chemistry. *Coordination chemistry reviews* 178 (1998), 699–721.
- [4] Carles Curutchet and Benedetta Mennucci. 2017. Quantum chemical studies of light harvesting. *Chemical reviews* 117, 2 (2017), 294–343.
- [5] Jack Deslippe, Georgy Samsonidze, David A Strubbe, Manish Jain, Marvin L Cohen, and Steven G Louie. 2012. BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures. *Computer Physics Communications* 183, 6 (2012), 1269–1289.
- [6] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim YarKhan, Maksims Abalenkovs, Negin Bagherpour, et al. 2019. PLASMA: Parallel linear algebra software for multicore using OpenMP. *ACM Transactions on Mathematical Software (TOMS)* 45, 2 (2019), 1–35.
- [7] 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.
- [8] Mark Gates, Ali Charara, Jakub Kurzak, Asim YarKhan, Mohammed Al Farhan, Dalal Sukkari, and Jack Dongarra. 2020. SLATE users' guide. (2020).
- [9] Mark Gates, Stanimire Tomov, and Jack Dongarra. 2018. Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs. *Parallel Comput.* 74 (2018), 3–18.
- [10] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.
- [11] Gaël Guennebaud, Benoit Jacob, et al. 2010. Eigen. URL: <http://eigen.tuxfamily.org> 3, 1 (2010).
- [12] Jürgen Hafner. 2008. Ab-initio simulations of materials using VASP: Density-functional theory and beyond. *Journal of computational chemistry* 29, 13 (2008), 2044–2078.
- [13] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. 2011. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.
- [14] Jiajun Li, Denis Golez, Giacomo Mazza, Andrew J Millis, Antoine Georges, and Martin Eckstein. 2020. Electromagnetic coupling in tight-binding models for strongly correlated light and matter. *Physical Review B* 101, 20 (2020), 205140.
- [15] Hatem Ltaief, Piotr Luszczek, Azzam Haidar, and Jack Dongarra. 2012. Solving the generalized symmetric eigenvalue problem using tile algorithms on multicore architectures. In *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 397–404.
- [16] 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*. IEEE, 944–955.
- [17] 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.
- [18] Françoise Tisseur and Jack Dongarra. 1999. A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures. *SIAM Journal on Scientific Computing* 20, 6 (1999), 2223–2236.
- [19] Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. 2011. MAGMA Users' Guide. *ICL, UTK (November 2009)* (2011).
- [20] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. 2005. GROMACS: fast, flexible, and free. *Journal of computational chemistry* 26, 16 (2005), 1701–1718.
- [21] Hansheng Wang, Zhekai Duan, Zitian Zhao, Siqi Wu, Saiqi Zheng, Qiao Li, Xu Jiang, and Shaoshuai Zhang. 2025. Improving Tridiagonalization Performance on GPU Architectures. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 469–480.
- [22] David S Watkins. 1982. Understanding the QR algorithm. *SIAM review* 24, 4 (1982), 427–440.
- [23] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
- [24] Victor Wen-zhe Yu, Jonathan Moussa, Pavel Kus, 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. *Computer Physics Communications* 262 (2021), 107808.
- [25] Shaoshuai Zhang, Ruchi Shah, Hiroyuki Ootomo, Rio Yokota, and Panruo Wu. 2023. Fast symmetric eigenvalue decomposition via wy representation on tensor core. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 301–312.