Improving Tridiagonalization Performance on GPU Architectures

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

Zhekai Duan
University of Edinburgh
s2085313@ed.ac.uk

Zitian Zhao
University of Electronic Science and Technology of China
zhaozitian@uestc.edu.cn

Siqi Wu
University of Electronic Science and Technology of China
siqi-wu@std.uestc.edu.cn

Saiqi Zheng
Xi'an Jiaotong-Liverpool University
Saiqi.Zheng21@student.xjtlu.edu.cn

Qiao Li, Xu Jiang, Shaoshuai Zhang
University of Electronic Science and Technology of China
{liqiao,jiangxu,szhang94}@uestc.edu.cn

Abstract

Tridiagonalization, which is a key step in symmetric eigenvalue decomposition (EVD), aims to convert a symmetric matrix to a tridiagonal form. In Nvidia's cuSOLVER library, the FP64 precision tridiagonalization process only reach 2.1 TFLOPs out of 67 TFLOPs on H100 GPU, and it consumes a significant portion of the elapsed time in the entire EVD process, accounting for over 97%. Thus, improving the tridiagonalization performance is crucial on accelerating EVD. In this paper, we analyze the reasons behind the suboptimal performance of tridiagonalization on GPU architectures, and we propose a new double blocking band reduction algorithm along with an implementation of GPU-based bulge chasing to improve the tridiagonalization performance. Through experimental evaluation, the proposed FP64 precision tridiagonalization method yields up to 19.6 TFLOPs which is 9.3x and 5.2x faster compared cuSOVLER and MAGMA, respectively.

CCS Concepts • **Mathematics of computing** → **Mathematical software performance**; **Solvers**; • **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → **Parallel algorithms**.

Keywords Eigenvalue Decomposition, GPGPU, Numerical Linear Algebra, HPC, Matrix Computation

1 Introduction

Tridiagonalization aims to reduce a symmetric matrix to a tridiagonal form:

$$A = Q^{-1} \times T \times Q, A \in \mathbb{R}^{n \times n},$$

where T is a tridiagonal matrix and Q is an orthogonal matrix. The tridiagonalization is a key step in symmetric EVD problems, and is regarded as a 'preconditioner' to reduce the execution time of the subsequent iterative methods such as QR algorithm [24] and divide and conquer method [20]. However, the tridiagonalization is not efficient on high performance GPU architectures. For instance, on H100 GPU, in terms of FP64 precision, cuSOVLER [1] can only achieve around 2 TFLOPs, while the theoretical peak performance is 67 TFLOPs. Furthermore, compared to the iterative methods, the EVD spends over 97% elapsed time on tridiagonalization when the matrix is large (Figure 4). Thus, to improve the entire EVD performance, accelerating tridiagonalization is essential and crucial.

There have been various attempts to optimize the tridiagonalization process on GPUs. The main idea is dividing the tridiagonalization into two stages: 1) successive band reduction (SBR) to reduce the matrix to a band form, and 2) bulge chasing (BC) to convert the band form to a tridiagonal matrix. This 2-stage tridiagonalization method involves more BLAS3 operations compared to direct tridiagonalization and has been proven to be faster than cuSOVLER [14, 16, 17]. Unfortunately, even the SOTA 2-stage tridiagonalization implemented in MAGMA [22] still shows suboptimal performance, reaching only up to 3.4 TFLOPs on the H100 GPU.

Therefore, in this paper, we first analyze the reasons behind the suboptimal performance of 2-stage tridiagonalization on GPUs. Generally speaking, the symmetric rank-2k update (`syr2k`) in SBR and the CPU-based BC process are the primary bottlenecks. Based on our experiments, the `syr2k` operation can only achieve around 20 TFLOPs on H100 GPU. Additionally, the BC process is executed on the CPU rather

[1] https://docs.nvidia.com/cuda/cusolver/index.html

than the GPU, which accounts for around 50% of the elapsed time of the 2-stage tridiagonalization.

To tackle the bottlenecks, we propose two optimized algorithms that significantly accelerate the 2-stage tridiagonalization. The first algorithm, called double-blocking band reduction (DBBR), uses two different blocksize, enabling the internal `syr2k` operations to reach up to 50 TFLOPs on H100 GPU. This approach also provides a smaller bandwidth, which can be utilized to accelerate the subsequent BC process. The second algorithm is BC on GPU. Although the BC process is typically not considered for optimization by hardware accelerators [10], we fully exploited the parallelism of the BC process and implemented it on the GPU. Even though it is memory-bound, this approach enabled the BC process to achieve superior performance, running 12.5x faster than MAGMA on GPUs. We consider the main contributions of this paper to be:

- We analyze the performance of the conventional tridiagonalization algorithm and identify the reasons and bottlenecks behind its suboptimal performance.
- We propose, implement and optimize two novel algorithms to address the bottlenecks and challenges in conventional tridiagonalization, including double-blocking band reduction and GPU-based bulge chasing, which significantly outperform MAGMA's corresponding routines.
- We conduct extensive experiments across various matrix sizes and GPUs to demonstrate the efficiency of the proposed tridiagonalization algorithm. Compared to cuSOLVER and MAGMA, the speedup reaches up to 9.3x and 5.2x, respectively. Additionally, we integrate MAGMA's divide-and-conquer routine for end-to-end eigenvalue decomposition, which also outperforms the SOTA EVD solvers.

The rest of the paper is organized as follows: Section 2 explains the basic concepts of 2-stage tridiagonalization.Section 3 analyzes the performance of conventional 2-stage tridiagonalization and points out the bottlenecks. Section 4 gives the algorithmic design and section 5 presents the optimization details. Section 6 evaluates our implementations and shows end-to-end EVD performance. Section 7 introduces related work, and section 8 draws a conclusion and depicts our future work.

2 Background

2.1 Householder Transformation

The Householder reflector is an orthogonal projection constructed from reflection against a hyperplane. It is useful to transform a given vector orthogonally to an axis. Specifically, given a vector x, the orthogonal matrix $H(v) = I - 2vv^T/(v^Tv)$ where $v = ||x||e_1 - x$ will map x to the first axis: $H(v)x = [||x||, 0, ..., 0]^T$. The above transformation is a rank-1 update, in other words, a BLAS2 operation. Fortunately, we can accumulate several Householder transformations by the WY representation [3] into a block to enrich BLAS3 operations. Suppose we have k Householder matrices $[H_1, H_2, ..., H_k]$, the WY representation will be:

$$H_1H_2...H_{k-1}H_k = I - W_kY_k^T$$

and if the $(k + 1)$-th block is factorized, then we have:

$$H_{k+1} = I - w_{k+1}y_{k+1}^T$$

$$Y_{k+1} = [Y_k|y_{k+1}]$$

$$W_{k+1} = [W_k|w_{k+1} - W_kY_K^Tw_{k+1}]$$

The tridiagonalization process, including the direct tridiagonalization used in cuSOLVER and the 2-stage tridiagonalization implemented in MAGMA [22], LAPACK [2], and PLASMA [7], requires the Householder transformation to eliminate off-diagonal and off-band entries.

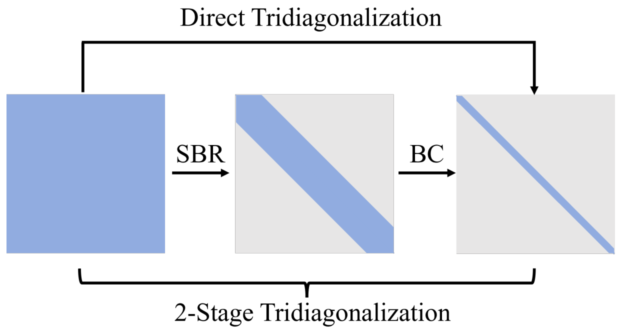


Figure 1. The difference between direct and 2-stage tridiagonalization

2.2 Tridiagonalization and 2-stage Tridiagonalization

The tridiagonalization process is usually the pre-step to eigenvalue decomposition. The goal of tridiagonalization can be expressed as follows:

$$A = Q^{-1} \times T \times Q, A \in \mathbb{R}^{n\times n},$$

where Q is an orthogonal matrix, and T is a tridiagonal matrix. To improve performance on modern computer architectures, Dongarra et al. [8] introduced blocked tridiagonalization. However, there are still approximately 50% unblocked BLAS2 operations. To further increase the number of BLAS3 operations, a 2-stage tridiagonalization method was proposed [14]. The first stage, known as SBR, converts the symmetric matrix to a band form, and the second BC stage reduces the band matrix to a tridiagonal matrix. The difference is illustrated in Figure 1.

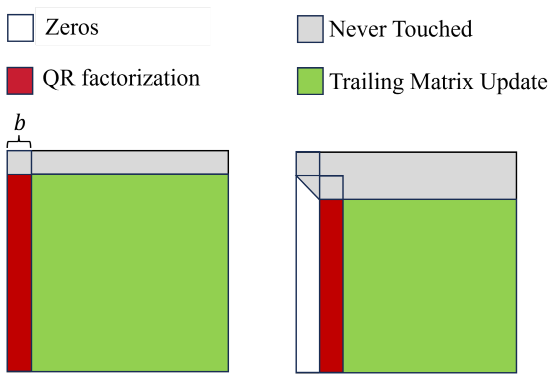


Figure 2. The first two iterations in SBR

2.3 Successive Band Reduction

Figure 2 illustrates the first two steps in SBR. The red block represents a panel, and we can QR factorize the panel such that $QR(\text{Panel}) = (I - WY^T)R$, eliminating the off-band elements. The R matrix will then overwrite the upper triangular part of the panel. The subsequent trailing matrix update is a two-sided operation using the ZY representation, which can leverage the `syr2k` routine [8], as shown in Equation 1.

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

Then we can treat the updated trailing matrix (green block) as a new full matrix and factorize the matrix iteratively.

2.4 Bulge Chasing

The BC process reduces a band matrix to a tridiagonal or bidiagonal matrix [9]. In essence, the steps of bulge chasing are quite similar to those of tridiagonalization, as both iteratively apply Householder transformations to the trailing matrix. The key difference is that the input matrix for BC is a band matrix, allowing the computations to exploit the band structure and reduce the number of operations.

The steps of one sweep in bulge chasing are illustrated in Figure 3. The process iteratively finds the Householder vectors and applies the Householder transformation to chase the bulge, which is generated by these transformations, until the bulge is swept out of the matrix. In Figure 3, the orange columns denote the search for Householder vectors to eliminate elements in the current column. Once the Householder vector v is formed, we use $H(v) = I - 2vv^T/(v^Tv)$ to update the diagonal block B_d, which is shown in red, from both the left and right sides such that $B_d = H(v)^{-1}B_dH(v)$. The off-band block B_{ol}, shown in green on the left side of B_d, is updated only from the left side, such that $B_{ol} = H(v)^{-1}B_{ol}$. The off-band block B_{od}, shown in green below B_d, is also updated from the left side, but in its transposed form, $B_{od}' = H(v)^{-1}B_{od}'$, which creates a bulge denoted by the blue blocks.

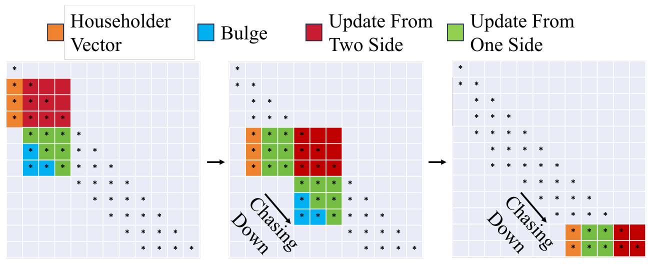


Figure 3. The single sweep of bulge chasing

To chase down the bulge, we find the Householder vectors of the first column of the bulge and repeat the above steps until the bulge is swept down to the last column.

The above steps demonstrate one sweep of the bulge chasing process. In fact, to fully reduce the band form matrix to a tridiagonal matrix, $n - 2$ sweeps are needed if the given matrix's size is $n \times n$.

3 Performance Analysis

3.1 An Overview of EVD Performance

As previously mentioned, the tridiagonalization process is a critical path in EVD in terms of performance. To substantiate this claim, we test several routines in cuSOLVER using FP64 precision, including tridiagonalization (`Dsytrd`) and EVD (`Dsyevd`), and in MAGMA, including SBR (`Dsy2sb`), the BC process (`Dsb2st`), and divide and conquer (`Dstedc`), using a matrix size of 49152×49152 (the maximum size supported by cuSOLVER on H100 GPU). The results are shown in Figure 4, that the overall EVD solver only spends 2.3% and 7.6% of the elapsed time on divide and conquer. This suggests that highly optimized tridiagonalization significantly contributes to overall performance. Unfortunately, in terms of FLOPs, the tridiagonalization process in cuSOLVER and MAGMA achieves only 2.0 TFLOPs and 3.4 TFLOPs on a matrix size of 49152 × 49152. This indicates that the SOTA tridiagonalization process utilizes only about 5% of the hardware's peak performance (67 TFLOPs).

Another observation from Figure 4 is that the BC process accounts for 48% of the elapsed time in the 2-stage tridiagonalization implemented in MAGMA, although the number of floating point operations is less than 10%. Despite this, the TFLOPs of SBR is around 9, indicating that the BC process is a critical path in 2-stage tridiagonalization.

By comparing the performance between cuSOLVER and MAGMA, and noting that the divide and conquer routine in MAGMA is slower than in cuSOLVER, we observe that the overall EVD performance in MAGMA, which uses 2-stage tridiagonalization, is superior to that in cuSOLVER. Therefore, we will focus on analyzing 2-stage tridiagonalization rather than direct tridiagonalization.

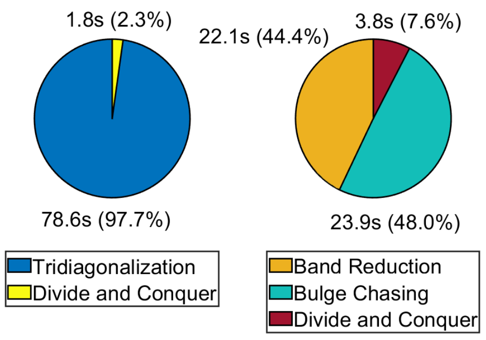
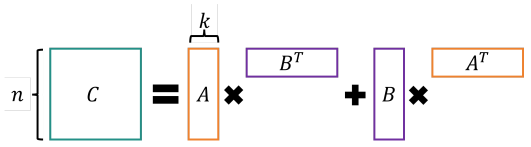


Figure 4. The elapsed time (in seconds) percentage of different operations in EVD with matrix size 49152 × 49152 on H100 GPU. The left pie chart shows cuSOVLER performance (tridiagonalization 2.0 TFLOPs); and the right pie chart shows the MAGMA 2-stage tridiagonalization and overall EVD performance (tridiagonalization 3.4 TFLOPs)



GPU k	H100 $n = 8192$	H100 $n = 32768$	RTX 4090 $n = 8192$	RTX 4090 $n = 32768$
16	0.43	3.58	1.07	1.19
32	0.86	7.02	1.07	1.20
64	1.71	12.78	1.06	1.21
128	3.39	21.05	1.06	1.21
256	6.41	30.13	1.12	1.22
512	11.57	38.31	1.20	1.24
1024	18.91	42.86	1.22	1.24
2048	27.21	45.36	1.23	1.24
4096	34.59	45.54	1.24	1.25

Table 1. The performance of SYR2K on H100 and RTX 4090 GPU with different input sizes (n and k) in TFLOPs

3.2 Performance Analysis on SBR

Refer to Figure 2, the SBR firstly selects a panel with bandwidth b and then QR factorizes factorizes the panel obtaining matrix Z and Y. Finally it updates the trailing matrix by `syr2k`.

The performance of `syr2k` is crucial to overall efficiency because the number of floating point operations in `syr2k` is much larger than that in panel QR factorization ($O(n^2b)$ vs. $O(nb^2)$, where n is the number of rows in the panel). We list the `Dsyr2k` performance using cuBLAS [2] on H100 and RTX 4090 GPUs in Table 1. These two GPUs represent emerging high performance GPUs and lower computing capacity GPUs, respectively. We observe that, given $k = 128$ (the typical selection in SBR), the `syr2k` performance on H100 is 21.05 TFLOPs, which is far from the peak. In contrast, the performance on RTX 4090 is close to the hardware's peak performance. This difference explains why SBR is efficient on older GPU architectures but not on emerging GPUs. Additionally, this discrepancy can be interpreted using the roofline model [25].

Thus, according to Table 1, on emerging GPU architectures such as H100, increasing the bandwidth b is a viable strategy to improve SBR performance. However, a larger b results in a significantly slower BC process. For example, given a matrix size of 49152 × 49152, setting $b = 64$ yields the performance shown in Figure 4 (SBR takes 22.1 seconds and BC takes 23.9 seconds). When b is increased to 128, SBR spends 16.5 seconds, but BC costs 84.9 seconds. As a result, the entire tridiagonalization process becomes slower, although the bandwidth is only increased from 64 to 128. Consequently, balancing the performance of SBR and BC is essential to ensure optimal tridiagonalization performance, which is why the bandwidth is typically selected to be less than 128.

[2]https://docs.nvidia.com/cuda/cublas/index.html

3.3 Performance Analysis on BC

The BC process is considered not to benefit from GPUs. Gates et.al. claims that the bulge chasing process, being limited in parallelism and close to memory bandwidth limits, would not benefit significantly from an accelerator-based implementation, as it is already optimized for CPU caches (Section 4.2 in [10]). So, the bulge chasing named `sb2st` is always implemented in numerical linear algebra libraries including LAPACK [2], PLASMA [7] and MAGMA [22] using CPUs.

The BC process is definitely memory-bound as b is typically very small. But there still remains a question whether it really lacks parallelism. According to Figure 3, the 4th Householder transformation in the i-th sweep does not have data dependency with the 1st Householder transformation in the $i+1$-th sweep. This observation leads to forming a pipeline of different sweeps and the pipeline is also leveraged in MAGMA [14]. However, it is unknown why there is no attempt of deploying BC on GPUs in previous studies.

To evaluate if the BC process can have better performance on GPU, we create a simple performance model to simulate the performance. Assume the matrix size is $n \times n$, the bandwidth is b. To simplify the calculation, we assume the bandwidth b can be divided by n. There are three laws in calculating the number of bulges:

① The $i+1$-th sweep starts after processing the first three bulges in the i-th sweep;

② The number of bulges decreases by one after b sweeps;

③ There will be a stall if the hardware is too busy to support an additional sweep.

If the hardware supports all of the tasks running in parallel in the pipeline, and assume the delay of each sweep is 3 bulges, then the total number of successive bulges is $3n-2$. The simplified estimation is based on the law ① and ②, that we can only count the blank bulges in front of each sweep ($3(n-1)$) and finally add the number of bulge of the last sweep (only one bulge). However, there're in total n sweeps to be processed and even the latest hardware cannot provide such number of cores if n is large.

Considering inductive law ③, suppose that the maximum number of sweeps supported in the pipeline is S. After S sweeps, the next sweep must wait for the first sweep to finish, causing a stall. The duration of each stall depends on the number of remaining bulges to be processed. After S sweeps, the first sweep has processed $3(S-1)$ bulges, leaving $\frac{n}{b}-3(S-1)$ bulges. There will be a total of $\frac{n}{S}$ stalls. Additionally, based on law ②, during the i-th stall, the number of stall cycles will be $\frac{n}{b}-\left\lfloor\frac{(i-1)S}{b}\right\rfloor-3(S-1)$. To simplify the analysis, let's omit the $\lfloor\rfloor$ symbol. We find that the stall cycles will be zero when $i \geq \frac{n+3b}{S}-3b+1$. Therefore, the total number of stall cycles is expressed as follows:

$$\sum_{i=1}^{\frac{n+3b}{S}-3b}\left(\frac{n+S}{b}-3S+3-\frac{S}{b}i\right).$$

The total cycles of BC is the sum of stall cycles and successive bulges cycles, it can be expressed:

$$3n-2+\sum_{i=1}^{\frac{n+3b}{S}-3b}\left(\frac{n+S}{b}-3S+3-\frac{S}{b}i\right).$$

Considering the parallelism from the SIMT instructions on GPU, our experimental results show that the approximate time for chasing down one bulge is around $10ms$ on H100 GPU. The total elapsed time for bulge chasing on the GPU can be evaluated by specifying n, S, and b. Figure 5 presents the estimated performance of GPU-based bulge chasing using the above performance model. The matrix size in Figure 5 is 65536×65536 with a bandwidth of $b = 32$, and S is varied from 1 to 128. The red dashed line in Figure 5 represents the MAGMA `sb2st` benchmark using the same n and b. From the figure, we observe that if the bulges are processed serially, the GPU-based version is significantly slower than MAGMA. However, if the number of parallel sweeps is 32 or more, the GPU-based bulge chasing can potentially outperform the MAGMA benchmark.

The question now is whether the GPU can provide sufficient parallelism to support more than 32 parallel sweeps. Recent GPUs, such as H100 [5], have more than 100 streaming multiprocessors (SMs). This implies that even if each SM processes only one sweep, the anticipated performance for bulge chasing could still surpass that of MAGMA. However, the performance model provides only a rough estimation, as other factors, such as cache reuse and synchronization between different sweeps, are not considered. Nevertheless, this at least demonstrates that the bulge chasing process has sufficient parallelism to be utilized effectively.

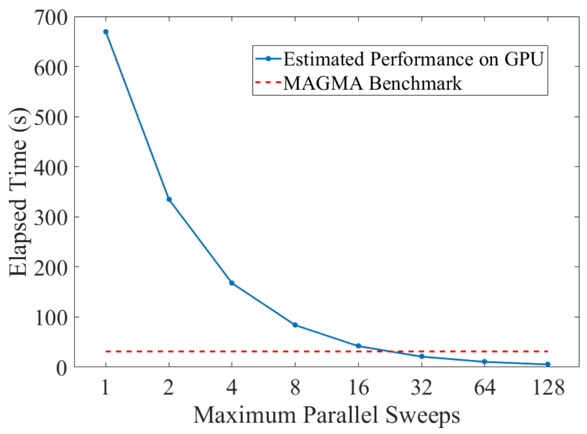


Figure 5. The estimated bulge chasing performance on GPU with matrix size 65536×65536 and different maximum allowed parallel sweeps

4 Algorithmic Design

4.1 Double Blocking Band Reduction

To improve the SBR performance, the key is to increase the dimension k. Based on the previous performance analysis, k equals b, and increasing k recklessly can lead to lower tridiagonalization performance. Therefore, we introduce a novel double blocking scheme for band reduction to increase k while maintaining a small b. Figure 6 illustrates the steps of the double blocking band reduction (DBBR). In DBBR, we set up two blocksize: the first block size is the same as in SBR, which equals the bandwidth b, while the second block size k is related to the dimension k, as shown in Table 1.

The first step is the same as SBR that QR factorizes the panel, while the trailing matrix update is different. When the working index is between b and k, we don't update the entire trailing matrix. Instead, we update the next panel until the working index reaching k. Finally, we update the trailing matrix entirely using `syr2k`. DBBR allows the k in `syr2k` to be larger and meanwhile keep small bandwidth b. Another advantage of DBBR is that smaller bandwidth is also allowed. For example, we can set the b to be 32, so that the following BC process can also be accelerated. For instance, with matrix size 49152×49152, BC process spends 16.2s and 23.9s in MAGMA when $b = 32$ and $b = 64$, respectively. See Algo 1 for more details.

4.2 Bulge Chasing on GPU

Based on the previous analysis, the BC process has the potential to be accelerated by GPUs if we can fully exploit the

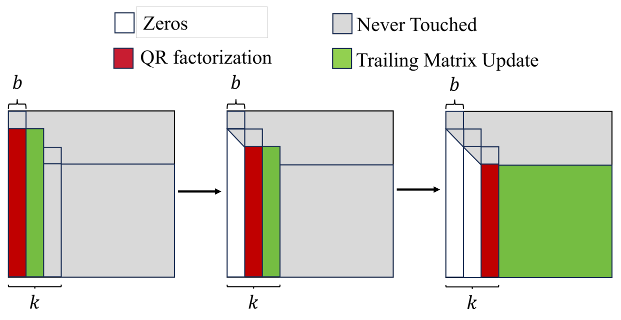


Figure 6. The steps of double blocking band reduction

Algorithm 1 Double Blocking Band Reduction

```
Input: A full matrix $A \in \mathbb{R}^{n \times n}$
Output: Perform an in-place band reduction on $A$ to a band
    form matrix $B \in \mathbb{R}^{n \times n}$ with bandwidth $b << n$.
1: for $i = 1 : k : n$ do
2:     % Accumulate the matrix Z
3:     for $j = i : b : i + k$ do
4:         % Perform QR decomposition on the red panel
5:         $[z, y] = PanelQR(A(j + b : n, j : j + b))$;
6:         $Z = [Z|z]; Y = [Y|y]$;
7:         % Update the green panel
8:         $PA = A(j + b : n, j + b : j + 2b)$;
9:         $PZ = Z(j + b : n, :)$;
10:        $PY = Y(j + b : n, :)$;
11:        $PA = PA - PZ * PY(1 : b, :)' - PY * PZ(1 : b, :)'$
12:        $A(j + b : n, j + b : j + 2b) = PA$;
13:    end for
14:    % Update the tail matrix with accumulated Z
15:    $A(i+k : n, i+k : n) = syr2k(A(i+k : n, i+k : n), Z, Y)$;
16: end for
```

parallelism within it. To gain a better understanding, we divide the parallelism into two groups: the parallelism within a single sweep and the parallelism between different sweeps.

the parallelism within a single sweep is straightforward as the GPUs provide SIMT instructions. As the BC process is iteratively performing Householder transformations, we can let multiple threads handle with one transformation in parallel. For example, line 11-13 shown in Algo 2 can be computed quickly using one warp which contains 32 threads.

The combination of different sweeps constitutes the entire bulge chasing process. We only launch $n - 2$ thread blocks, and each thread block handles one bugle-chasing sweep using the kernel described in Algorithm 2. The only concern is determining how to synchronize those kernels. As analyzed previously, when the i-th sweep is handling the 4th Householder transformation, the $(i + 1)$-th sweep can start safely. However, since the GPU cannot guarantee that the i-th sweep will always complete 3 Householder transformations ahead, we must set up a lock to prevent the sweeps from being executed out of order. This lock is implemented using a volatile array and a while loop, as Algorithm 2 shows. In short, the kernel for the $(i + 1)$-th sweep continuously observes the working row of the i-th sweep and spins until there is no data dependency between the i-th and $(i + 1)$-th kernels.

Algorithm 2 One single sweep in bulge chasing

```
Input: A band form matrix $B \in \mathbb{R}^{n \times n}$ with bandwidth $b <<$
    $n$ and $i$ is the sweep index
Output: Sweep down the $i$-th column to the right-bottom
1: volatile int $gCom[n] = 0$;
2: % Setup a flag $f_i$ denotes the working row
3: $gCom[i] = i + 1$
4: for $j = i + 1 : b : n$ do
5:     while $((0! = i)$ && $(gCom[i] + 2 * b > gCom[i - 1]))$
       do
6:         Continue
7:     end while
8:     % Compute Householder vectors $v_i$
9:     $H(v_i) = I - 2\frac{v_i v_i^T}{v_i^T v_i}$
10:    % Use $v_i$ to chase the bugle
11:    $B_d = H(v_i)^{-1} B_d H(v_i)$ % $B_d$ is the diagonal block
12:    $B_{ol} = H(v_i)^{-1} B_{ol}$ % $B_o$ is the off-band block on the
       left side
13:    $B_{od}' = H(v_i)^{-1} B_{od}'$ % $B_o$ is the off-band block on the
       down side
14:    $gCom[i] = j + b$ % Update the flag $f_i$
15: end for
```

4.3 Back Transformation

Because of the same reason explained in Section 3.2, the back transformation in SBR also suffers from low efficiency in GEMMs. Assuming the orthogonal matrix Q_1 has already been obtained from bulge chasing, the entire orthogonal matrix Q is formed by:

$$Q = Q_1 \times (I - W_1 Y_1^T) \times (I - W_2 Y_2^T) \times \cdots \times (I - W_n Y_n^T),$$

where W_i and Y_i are already formed by the QR factorization in SBR. This back transformation process typically uses the `ormqr` routine defined in LAPACK in the following order: $Q = Q_1 \times (I - W_1 Y_1^T)$, $Q = Q \times (I - W_2 Y_2^T)$, ..., $Q = Q \times (I - W_n Y_n^T)$. Thus, the GEMMs in the back transformation also always have dimensions equal to the bandwidth b. To address this issue, we can consider combining successive W matrices recursively, as shown in Algorithm 3 (line 12). Specifically, $W_{k+1} = [W_k \mid w_{k+1} - W_k Y_k^T w_{k+1}]$. The advantage of this recursive algorithm is that it can produce much larger and more square GEMMs, which can benefit more from modern GPUs. However, the disadvantage is that the recursive algorithm requires forming the entire W matrix, whereas the conventional back transformation does not. Consequently, the number of floating point operations also increases. We will discuss optimization techniques in the following section and evaluate whether the recursive algorithm can achieve speedup.

Algorithm 3 Back transformation in DBBR

```
Input: Blocks of $W$ and $Y$ and orthogonal matrix $Q$
Output: The eigenvector $Q$
    function $[W]$ = ComputeW$(W, Y)$
1:  $n = length(W)$;
2:  if $n <= 2 * b$
3:      $W_1 = W(:, 1 : b)$;
4:      $W_2 = W(:, b + 1 : 2 * b)$;
5:      $Y_1 = Y(:, 1 : b)$;
6:      $W = [W_1 | W_2 - W_1 * Y_1' * W_2]$;
7:  endif
8:  % left recurse
9:  $W_1 = ComputeW(W(:, 1 : n/2), Y(:, 1 : n/2))$;
10: % right recurse
11: $W_2 = ComputeW(W(:, n/2 + 1 : n), Y(:, n/2 + 1 : n))$;
12: $W = [W_1 | W_2 - W_1 * Y_1' * W_2]$;
13: $Q = (I - WY^T) * Q$
```

5 Optimization

5.1 Optimizations on DBBR

One of the bottlenecks of DBBR is the suboptimal performance of syr2k using cuBLAS. Firstly, cuBLAS syr2k exhibits a performance degradation issue when the matrix size is large. Secondly, even for smaller matrix sizes, syr2k can achieve less than 50 TFLOPs on the H100 GPU, which is significantly below the theoretical performance. To address these issues, we designed our own syr2k routine, and the steps are illustrated in Figure 7. Suppose we are computing a matrix $A = A - ZY^T - YZ^T$ with 4×4 blocks. In the first iteration, we compute the diagonal 4 blocks, as shown in the left graph of Figure 7. Next, we compute the two off-diagonal blocks, as illustrated in the middle graph. After two iterations, the resulting matrix is relatively more square compared to the conventional blocking syr2k algorithm [23]. This GEMM shape is potentially more efficient on emerging GPU architectures [27]. Additionally, note that all computations in Figure 7 are independent, allowing us to reorder computations to hide latency and utilize idle GPU resources more effectively.

To evaluate the performance, we compare the proposed syr2k implementation with the cuBLAS Dsyr2k routine. The results are shown in Figure 8. The proposed syr2k implementation outperforms cuBLAS Dsyr2k across various matrix sizes. Notably, when $n \geq 49152$, the performance of the cuBLAS syr2k routine drops significantly, while the performance of the proposed syr2k remains stable.

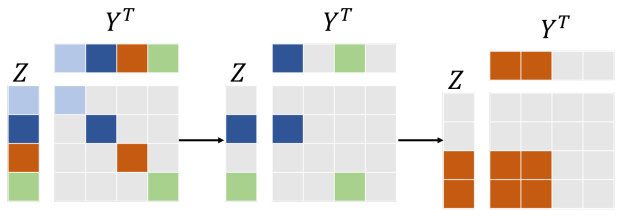


Figure 7. The steps of computing SYR2K

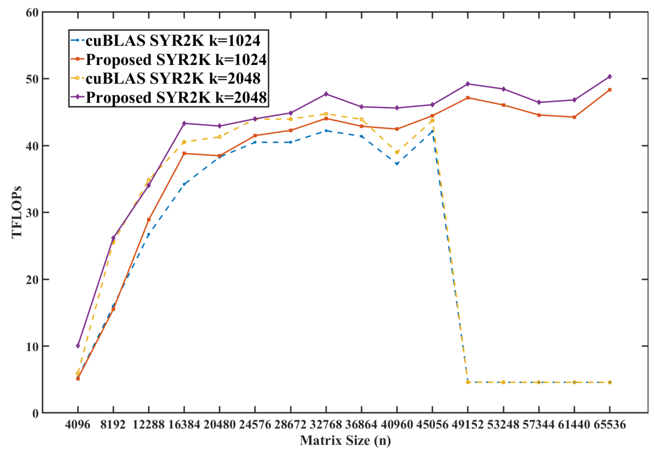


Figure 8. The performance comparison between the proposed SYR2K and cuBLAS SYR2K on H100 GPU

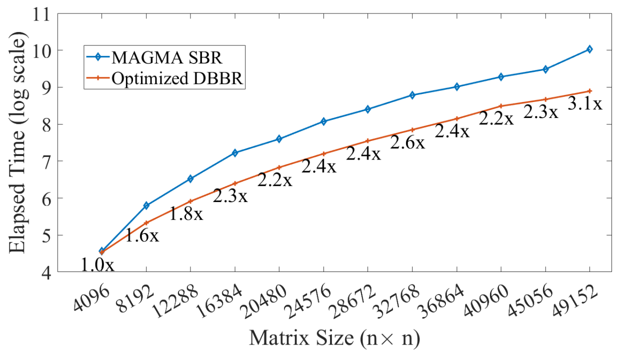


Figure 9. The band reduction performance comparison between MAGMA SBR and proposed DBBR with bandwidth $b = 64$ on H100 GPU

Integrating SYR2K into DBBR enhances its performance, particularly for large matrix sizes. To ensure a fair comparison, we restrict the matrix size to be less than 49152; otherwise, the low efficiency of the cuBLAS syr2k routine would overshadow MAGMA's SBR performance. As shown in Figure 9, DBBR outperforms SBR, especially for large matrix sizes, with an acceleration ratio of up to 3.1x.

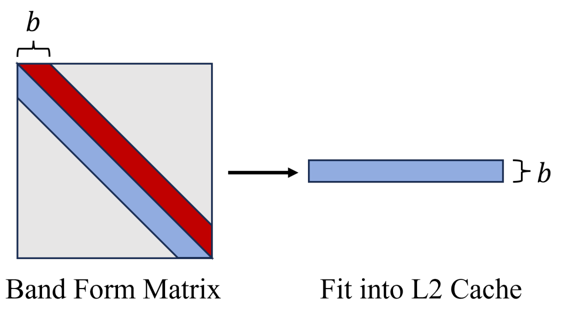


Figure 10. The explanation of how to reuse L2 cache

5.2 Optimizations on Bulge Chasing

Based on our performance analysis of BC on GPUs, BC is expected to be faster on GPUs than on CPUs due to the GPU's ability to provide sufficient parallelism for supporting asynchronous kernels during successive sweeps. To implement the GPU-based BC process, a straightforward approach involves assigning each threadblock to handle one sweep using Algorithm 2. Given that modern GPUs offer enough number of Streaming Multiprocessors (SMs), even this basic implementation can outperform the CPU-based BC. However, this naive approach incurs significant overhead due to frequent data movement between global memory and the L2/L1 cache, indicating potential for further optimization by improving cache reuse.

To enhance L2 cache efficiency, we employ a simple strategy illustrated in Figure 10. The entries of the original band matrix are nonconsecutive in memory, leading to numerous L2 cache misses. By storing these entries (taking advantage of the matrix's symmetric property) in a different memory space, we achieve consecutive memory access. Furthermore, since the L2 cache on GPUs like H100 is relatively large (50MB), nearly all elements can be stored within the L2 cache, thereby reducing the need for expensive global memory access.

For L1 cache or shared memory, the naive implementation, which uses one threadblock to handle one sweep, cannot take advantage of the L1 cache. This is because the data between successive sweeps can be also reused. Thus, we can group several sweeps together and make one warp instead of one threadblock to process one sweep. In addition, extra warp can also be invoked for prefetching to hide the data movement between L2 cache and L1 cache.

We compare the GPU-based BC performance with MAGMA in Figure 11, and we can find that the naive implementation can also have up to 5.9x speedup compared to MAGMA, which conforms to the performance model (Section 3.3, Figure 5). With the above optimization techniques, the performance can be even faster whose speedup is 12.5x when the matrix size is large. Consequently, the BC process is no longer a bottleneck (refer to Figure 4).

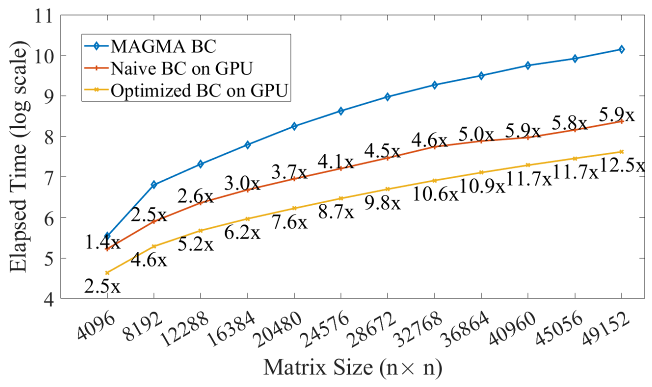


Figure 11. The performance comparison of BC between MAGMA sb2st routine, the naive GPU-based BC and optimized GPU-based BC on H100 GPU

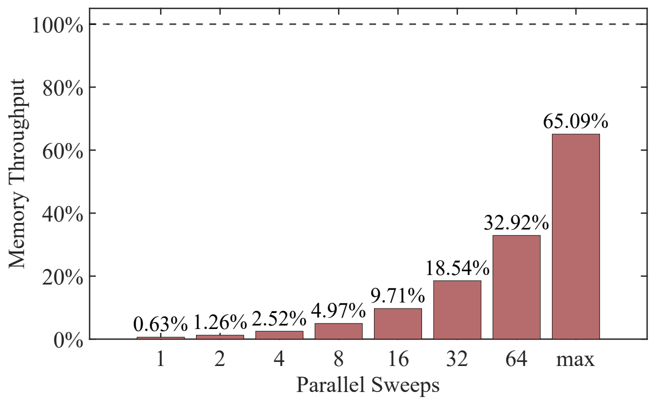


Figure 12. The memory throughput of bulge chasing using different number of parallel sweeps on H100 GPU; the max in the X-axis means the max sweeps supported by the hardware.

Furthermore, to analyze the hardware utilization in BC, we measured the memory throughput using Nsight Compute [3], with the results presented in Figure 12. Our findings demonstrate that increasing the number of parallel sweeps in BC significantly enhances memory throughput. In other words, greater parallelism in BC leads to higher memory bandwidth utilization, thereby improving performance on GPU architectures.

5.3 Optimization on Back Transformation

Although Algo 3 generates relatively square GEMMs, which achieve higher TFLOPs on modern GPUs, the additional computations required for the W matrix and the reduced parallelism limit the performance of the back transformation. To address this issue, we avoid forming the entire W matrix. Instead, we generate several blocks of W incrementally, ensuring that the dimension k is sufficiently large to maintain optimal performance.

[3]https://developer.nvidia.com/nsight-compute

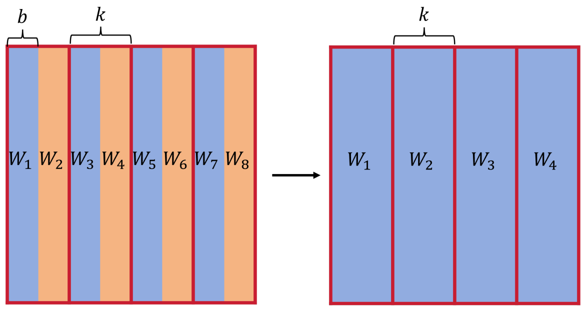


Figure 13. The steps of back transformation in DBBR

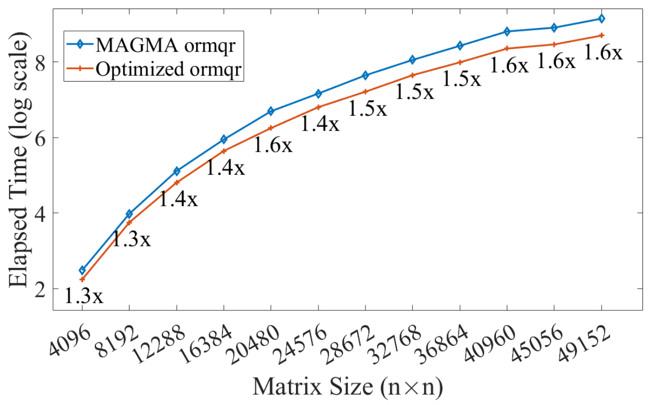


Figure 14. The performance comparison between MAGMA and the proposed back transformation routine in band reduction with bandwidth $b = 64$ on H100 GPU

Figure 13 explains the idea. Suppose the matrix size is $n \times n$, we have n/b blocks of $W \in \mathbb{R}^{n\times b}$ initially. Then we call batched GEMM to form the larger $W_i = [W_i | W_{i+1} - W_i Y_i^T W_{i+1}]$ in parallel recursively, until the size of W_i grows to $n \times k$. Afterwards, we follow the conventional ormqr routine to multiply the orthogonal matrix generated from the BC back transformation.

Figure 14 shows the performance comparison between MAGMA and the proposed back transformation in band reduction (ormqr). We set the bandwidth b to 64, because MAGMA can have the best performance. In terms of the elapsed time, even though the proposed method performs extra computations on forming unnecessary W matrices, the enlarged dimension k (we select $k = 2048$ in Figure 13) still brings satisfied accelerations across various matrix sizes.

6 Experimental Evaluation

We conducted experiments on a system running a 5.4.0-99-generic Linux operating system with NVIDIA H100-SMX and RTX 4090 GPU. The CUDA version used is 12.5, which includes a C++ compiler and the cuBLAS and cuSOLVER libraries. This section will primarily showcase the performance of the proposed tridiagonalization and the entire EVD on various GPU architectures.

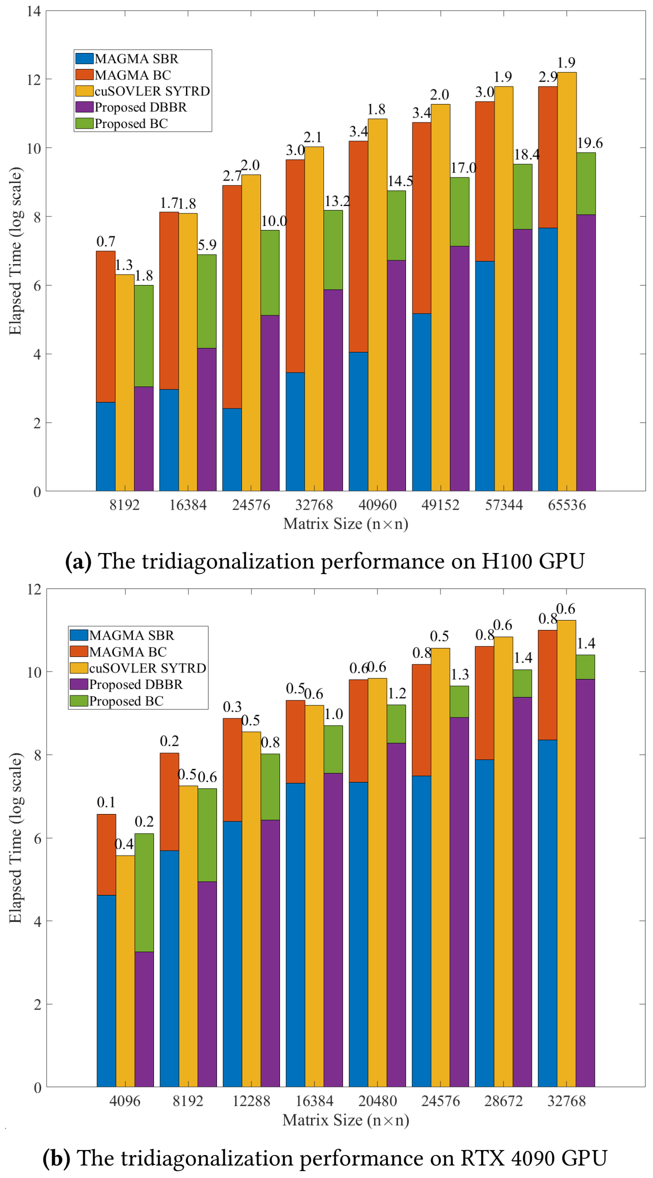


Figure 15. Tridiagonalization performance comparison among cuSOLVER, MAGMA, and the proposed method for different matrix sizes. The numbers on top of the bars denote TFLOPs, with the peak FP64 performance of the H100, and RTX 4090 GPUs being 67 TFLOPs and 1.29 TFLOPs, respectively

6.1 Tridiagonalization Performance

Figure 15 compares the tridiagonalization performance between MAGMA (`sy2sb` and `sb2st` routines), cuSOLVER (`sytrd`), and our proposed method (DBBR and GPU-based

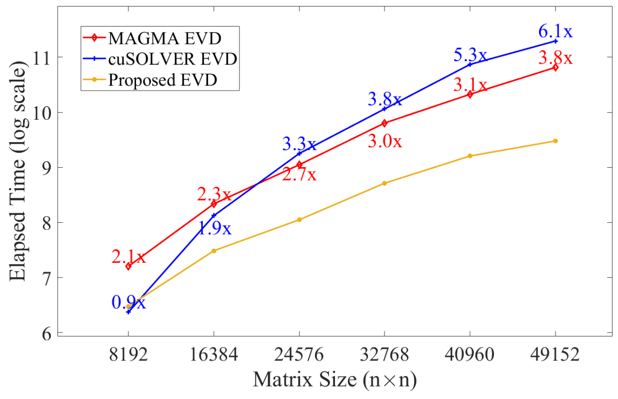
BC) on H100 and RTX 4090 GPUs. To achieve optimal performance, we set $b = 64$ and use 8 MKL threads for MAGMA, while $b = 32$ and $k = 1024$ were selected for our proposed method. On H100 GPU (Figure 15a), our method consistently outperforms cuSOLVER across all tested matrix sizes, whereas MAGMA's 2-stage tridiagonalization demonstrates superior performance only for large matrices. In terms of computational throughput, our method reaches 19.6 TFLOPs, significantly outpacing the performance of MAGMA and cuSOLVER, which are 3.4 TFLOPs and 2.1 TFLOPs, respectively. These results underscore the superior performance of our tridiagonalization approach on GPUs with high computational capacity.

We also conducted experiments on the RTX 4090 GPU, yielding two key insights. Firstly, our method achieves a maximum of 1.4 TFLOPs, slightly exceeding the RTX 4090's peak performance of 1.29 TFLOPs. This result is attributable to the acceleration provided by INT8 Tensor Cores [19], which neither MAGMA nor cuSOLVER can utilize. Secondly, despite the relatively low double precision computing capacity of the RTX 4090, our GPU-based BC significantly outperforms MAGMA. The elapsed time, presented on a log scale (with the bar height representing the ratio of SBR/DBBR to BC in the tridiagonalization process), reveals that BC takes 213ms vs. 209ms on a 4096×4096 matrix, and 14327ms vs. 1839ms on a 32768×32768 matrix for MAGMA and our method, respectively. This comparison highlights that BC performance is more dependent on parallelism than on computing capacity.

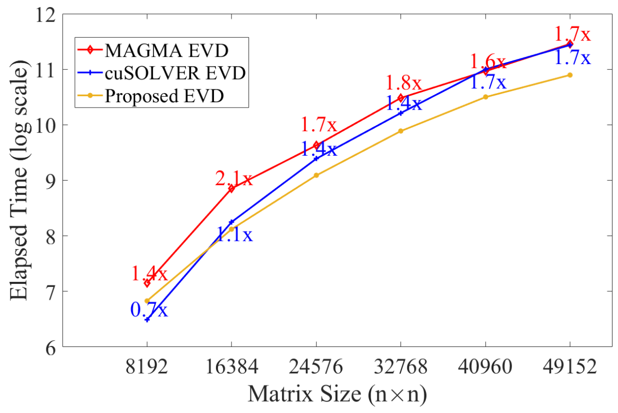
6.2 End-to-End EVD performance

We integrate the divide and conquer method as implemented in MAGMA to complete the entire EVD process. The speedups of our proposed EVD method compared to cuSOLVER and MAGMA are represented by the numbers along the blue and red lines in Figure 16, respectively. When eigenvectors are not required, our approach outperforms cuSOLVER, except for matrix sizes smaller than 8192. This is because the divide and conquer algorithm in cuSOLVER takes approximately 33 ms, whereas MAGMA requires 248 ms. For larger matrices, our method delivers speedups of up to 6.1x and 3.8x over cuSOLVER and MAGMA, respectively.

When eigenvectors are required, the limitations of the 2-stage tridiagonalization become apparent. Compared to direct tridiagonalization, the back transformation process in the BC process is significantly slower, consuming 61% of the total elapsed time in our proposed EVD process and 36% in MAGMA's process for a matrix size of 49152×49152. Despite this drawback, our proposed tridiagonalization and back transformation optimizations in DBBR enable our EVD approach to maintain a slight performance advantage over cuSOLVER for matrices larger than 8192.



(a) The EVD performance without eigenvectors



(b) The EVD performance with eigenvectors

Figure 16. EVD performance comparison among cuSOLVER, MAGMA, and the proposed method. The numbers along with the lines denote the speedups of our method compared to cuSOVLER and MAGMA

7 Related Work

7.1 Tridiagonalization

Tridiagonalization is commonly performed using Householder transformations [12]. To enhance execution efficiency on modern high-performance architectures, the WY representation technique [3, 21] is often applied during the transformation process. To further improve data locality, 2-stage tridiagonalization [14] is frequently employed for larger matrices and this method has been demonstrated to be highly efficient on multi-core architectures [10, 16, 17]. Zhang et al. [28] EVD using a 2-stage tridiagonalization approach on FP16 Tensor Cores. However, their work focuses on computing eigenvalues only and employs a low-precision SBR implementation using the WY representation. Besides, their claim regarding the ZY representation is incorrect. Compared to our proposed method, their approach achieves low speedup and has limited applicability.

7.2 Symmetric Eigenvalue Decomposition

In this paper, we mainly discuss the optimization on large tridiagonalization and EVD problem, and it has many applications among different disciplines including principle component analysis [1], low rank approximation [29], tight binding in condensed matter physics [15], quantum chemistry [6] and density function problems [4]. Solving large dense eigenvalue problems in computational science often involves using the ELPA library [4], which offers efficient solutions for large matrices on distributed systems equipped with GPU accelerators [18, 26].

The typically solution of EVD is tridiagonalization following an iterative method. QR algorithm [24], divide and conquer [13], and Jacobi iterations [11] are the most popular iterative methods. Among these, the divide and conquer method is particularly popular due to its superior parallelism and efficiency in computing eigenvectors. For computing eigenvalues alone, the QR algorithm is often the best choice. These methods are implemented in most linear algebra packages, including LAPACK [2], MAGMA [22], and cuSOLVER.

8 Conclusion and Future Work

In this paper, we introduce a novel 2-stage tridiagonalization algorithm that significantly outperforms the corresponding routines in cuSOLVER and MAGMA. We begin by analyzing the performance limitations of conventional 2-stage tridiagonalization, identifying the SBR and BC as critical bottlenecks. Specifically, in the SBR process, the trailing matrix update using `syr2k` underutilizes the computational capacity of modern GPUs. To solve this, we propose a double blocking band reduction algorithm, which substantially enhances performance. Additionally, we demonstrate that the BC process, traditionally CPU-bound, can be efficiently accelerated on GPUs, and we implement a GPU-based BC to further boost performance. Our experimental results show that the proposed tridiagonalization algorithm achieves up to 19.6 TFLOPs on H100 GPU, in contrast to the 2.1 TFLOPs and 3.4 TFLOPs attained by cuSOLVER and MAGMA, respectively.

We also evaluate the end-to-end EVD process using the divide and conquer algorithm as implemented in MAGMA. Although MAGMA's EVD implementation is nearly 6x slower than cuSOLVER, our approach delivers superior overall performance, achieving speedups of up to 6.1x and 1.8x, depending on whether eigenvectors are required. Besides, we optimize the bank transformation in DBBR, yielding a further 1.6x speedup over MAGMA's `ormqr` routine.

However, when eigenvectors are needed, the proposed method doesn't offer substantial speedup relative to cuSOLVER. This limitation is primarily due to the back transformation in BC, which dominates the computational time, spending 61% of the total execution time. Future work will focus on optimizing this back transformation process.

[4] https://elpa.mpcdf.mpg.de/ABOUT_ELPA.html

Acknowledgments

This work is supported by University of Electronic Science and Technology of China startup grant No. A1098531023601465 and the CCF-SUMA foundation grant No. CCF-GH OF 2024002. We are very grateful for the valuable suggestions from the anonymous reviewers and their time and efforts in reviewing the manuscript. Shaoshuai Zhang is the corresponding author.

References

[1] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[2] 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.

[3] Christian Bischof and Charles Van Loan. 1987. The WY representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput.* 8, 1 (1987), s2–s13.

[4] H Chermette. 1998. Density functional theory: a powerful tool for theoretical studies in coordination chemistry. *Coordination chemistry reviews* 178 (1998), 699–721.

[5] Jack Choquette. 2023. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro* (2023).

[6] Carles Curutchet and Benedetta Mennucci. 2017. Quantum chemical studies of light harvesting. *Chemical reviews* 117, 2 (2017), 294–343.

[7] 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.

[8] 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.

[9] Sebastian Gant. [n.d.]. Chasing the Bulge. ([n. d.]).

[10] 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.

[11] Gene H Golub and Henk A Van der Vorst. 2000. Eigenvalue computation in the 20th century. *J. Comput. Appl. Math.* 123, 1-2 (2000), 35–65.

[12] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations.* JHU press.

[13] Ming Gu and Stanley C Eisenstat. 1995. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.* 16, 1 (1995), 172–191.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Hiroyuki Ootomo, Katsuhisa Ozaki, and Rio Yokota. 2024. DGEMM on integer matrix multiplication unit. *The International Journal of High Performance Computing Applications* (2024), 10943420241239588.

[20] Jeffery D Rutter. 1991. A Serial Implementation of Cuppen's Divide and Conquer Algorithm.

[21] 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.

[22] Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. 2011. MAGMA Users' Guide. *ICL, UTK (November 2009)* (2011).

[23] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. 2016. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–11.

[24] David S Watkins. 1982. Understanding the QR algorithm. *SIAM review* 24, 4 (1982), 427–440.

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

[26] Victor Wen-zhe Yu, Jonathan Moussa, Pavel Ks, 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.

[27] Shaoshuai Zhang, Vivek Karihaloo, and Panruo Wu. 2020. Basic Linear Algebra Operations on TensorCore GPU. In *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 44–52.

[28] 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.

[29] Shaoshuai Zhang, Ruchi Shah, and Panruo Wu. 2020. TensorSVM: accelerating kernel machines with tensor engine. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–11.