W-Cycle SVD: A Multilevel Algorithm for Batched SVD on GPUs

Junmin Xiao
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
xiaojunmin@ict.ac.cn

Yunfei Pang
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
pangyunfei20g@ict.ac.cn

Qing Xue
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
xueqing@ncic.ac.cn

Chaoyang Shui
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
shuichaoyang@ncic.ac.cn

Ke Meng
Alibaba Group
Beijing, China
septic.mk@gmail.com

Hui Ma
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
mahui@ncic.ac.cn

Mingyi Li
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
limingyi@ncic.ac.cn

Xiaoyang Zhang
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
zhangxiaoyang@ncic.ac.cn

Guangming Tan
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
tgm@ict.ac.cn

***Abstract*—As a basic matrix factorization operation, Singular Value Decomposition (SVD) is widely used in diverse domains. In real-world applications, the computational bottleneck of matrix factorization is on small matrices, and many GPU-accelerated batched SVD algorithms have been developed recently for higher performance. However, these algorithms failed to achieve both high data locality and convergence speed, because they are size-sensitive. In this work, we propose a novel W-cycle SVD to accelerate the batched one-sided Jacobi SVD on GPUs. The W-cycle SVD, which is size-oblivious, successfully exploits the data reuse and ensures the optimal convergence speed for batched SVD. Further, we present the efficient batched kernel design, and propose a tailoring strategy based on auto-tuning to improve the batched matrix multiplication in SVDs. The evaluation demonstrates that the proposed algorithm achieves 2.6∼10.2× speedup over the state-of-the-art cuSOLVER. In a real-world data assimilation application, our algorithm achieves 2.73∼3.09× speedup compared with MAGMA.**

***Index Terms*—Singular Value Decomposition, GPU**

I. Introduction

The Singular Value Decomposition (SVD) is to factorize a matrix A into the form $A = U\Sigma V^T$. It is a basic matrix factorization, which generalizes the eigenvalue decomposition (EVD) of a positive semi-definite matrix [1]. Many real-world applications in diverse domains [2]–[4], such as scientific computing, machine learning, and image processing, deeply depend on this kernel. For example, SVD enables us to keep the primary singular values of an image for retaining the image quality in data compression and reconstruction. Unfortunately, the real-world data processing problems usually involve a large number of small-matrix SVDs (with the numbers of rows and columns not larger than $1,024$), which is time-consuming and has been the bottleneck of the whole workflow [2], [3].

To implement a high-performance SVD kernel, various methods have been developed [1], among which QR-based [5]–[7] and Jacobi-based [8] algorithms are most commonly used. The QR-based algorithms are faster than the Jacobi-based algorithms in sequential computing [6]. However, the Jacobi-based algorithms provide singular values as well as left and right singular vectors with higher relative accuracy [9], [10], which is crucial to many applications.

The Jacobi-based algorithms have recently attracted a lot of attention [11]–[13], including one-sided and two-sided Jacobi. Comparatively speaking, the two-sided Jacobi method usually converges fast on symmetric matrices, but it is computationally expensive and inapplicable to vector pipeline computing. Thus, the one-sided Jacobi method is widely adopted to achieve high performance for parallel SVD [14]–[18], which motivates the recent study to accelerate the implementation [19] and develop a variety of optimizations [20]–[22]. The fundamental difference between one-sided and two-sided Jacobi methods is in the sub-matrix orthogonalization. The two-sided Jacobi method orthogonalizes rows and columns of a matrix at the same time, which requires a sequential workflow, while the one-sided Jacobi method only performs the orthogonalization of columns so that the individual pairs of columns can be orthogonalized in parallel, which provides more opportunities to achieve high performance for SVD.

In the one-sided Jacobi method, a matrix A is partitioned into pairs of column blocks. All the singular values of A are

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

obtained until all pairs of column blocks are orthogonalized with each other. The orthogonalization of column block pairs is usually assigned to thread blocks (or warps) in Graphic Processing Unit (GPU), which involves the General Matrix Multiplication (GEMM) of column block matrices and the inner product of column vectors. In addition, GPUs have been demonstrated to be able to provide tremendous computation power for accelerating regular applications such as GEMMs [23]–[25], which is of significant importance in SVD.

As motioned above, real-world applications usually involve a large number of small-matrix SVDs, thus it is essential to adopt batched SVD to improve throughput and resource utilization on GPUs. The recent work has proposed optimizations for kernel functions of batched one-sided Jacobi method [19]–[21]. For example, NVIDIA provides a batched one-sided Jacobi SVD (cublasSVDBatched) in cuSOLVER [20], which applies to the matrices with the numbers of rows and columns smaller than 32. The state-of-the-art batched algorithms on GPUs were proposed to accelerate the orthogonalization of column blocks in SVDs with the same matrix size [19].

In real-world cases, the matrix sizes in a batched SVD task may vary significantly, thus the optimal algorithm for a single SVD does not suit the batched one. A fine-grained analysis [19] on batched SVD suggests that diverse algorithm designs for matrices of diverse sizes are necessary to achieve high performance on GPUs. However, the recent work uses a uniform column block width to provide ''one-size-fit-all'' solution for all the SVDs in a batched task, which is static for diverse matrix sizes. These algorithms can only achieve sub-optimal for each SVD in a batched task since the high data reuse and convergence speed can hardly be achieved simultaneously.

To address this diverse size problem for the batched SVD, we review the implementations of one-sided Jacobi method for matrices of different sizes, and find that the different implementations can be described using a uniform multilevel workflow with respect to recursion. Based on the insight that matrices of different sizes require different algorithm designs, we propose a novel algorithm called W-cycle SVD, which supports a uniform workflow for the batched one-sided Jacobi SVD with diverse matrix sizes. Furthermore, two kernels are designed to accelerate SVD and EVD processes in our algorithm. Meanwhile, a tailoring optimization is also developed to improve the parallelism and data reuse of batched GEMM in SVDs.

W-cycle SVD is a novel multilevel algorithm that performs different column block rotations for a batched SVD at different levels, which features the following that differs from previous work. (1) It ensures that all the decomposed matrices at any level can be stored entirely in GPU shared memory to exploit the data reuse of on-chip memory. (2) It selects a specific optimization parameter for each matrix to ensure the optimal convergence speed of batched SVD. (3) In prior work, the multi-grid methods involving a W-cycle workflow were developed for eigenvalue problems [26], [27]. In these methods, coarse grids are actually smaller dimension, and grid transfers are interpretable in terms of accuracy and aliasing. However, in the multilevel workflow of W-cycle SVD, lower levels involves smaller tiles, and level transfers are splitting tiles further. Specifically, our contributions can be summarized as follows:

- Propose a W-cycle SVD algorithm based on an in-depth analysis on the performance of batched SVD and two attractive observations. The proposed algorithm successfully exploits the data reuse and ensures the optimal convergence speed for batched SVD on GPUs (Section III).
- Design two efficient batched kernels for the SVD and EVD processes within GPU shared memory respectively, and develop a tailoring strategy to accelerate batched GEMM in SVD by fully exploiting the parallelism and data reuse, which further improves the performance of the W-cycle SVD algorithm (Section IV).
- Evaluate and analyze the performance of the W-cycle SVD, and prove that the proposed algorithm supports the batched SVD on GPUs well (Section V).

II. Background

A. Basic Idea of One-sided Jacobi Method for SVD

Assume that A is a $m \times n$ matrix. The SVD of A is to factorize A into the form

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

where both $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are unit orthogonal matrices, and $\Sigma \in \mathbb{R}^{m\times n}$ is a nonnegative diagonal matrix whose main diagonal arranges all the singular values of A. Meanwhile, U and V are called the left and right singular matrices of A respectively.

For SVD computation, the one-sided Jacobi method has recently attracted a lot of attention because of its intrinsic parallelism [14]–[22]. To achieve high parallel performance, the one-sided Jacobi method applies plane rotation matrices $J^1, J^2, \cdots, J^k$ on the right side of A to orthogonalize the columns of A. All the columns of A converge to $U\Sigma$, that is,

$$AJ^1J^2\cdots J^k \to U\Sigma, \quad \text{as} \quad k \to \infty. \tag{2}$$

By accumulating the rotations, we have $V = J^1J^2\cdots$, which gives all the right singular vectors. Each rotation matrix J^k is determined by orthogonalizing any pair of A's column blocks.

B. One-sided Jacobi Method using Column Block Rotations

Algorithm 1 shows the one-sided Jacobi method based on column block rotations. For a given width w of column blocks ($1 < w \leq n/2$), the matrix A can be rewritten as a column block form, i.e., $A = [A_1, \cdots, A_{n/w}]$ where $A_i \in \mathbb{R}^{m\times w}$. To orthogonalize two column blocks A_i and A_j, the one-sided Jacobi method joins A_i and A_j together to form $A_{ij} = [A_i, A_j] \in \mathbb{R}^{m\times 2w}$, and updates $A_i = \hat{A}_i$ and $A_j = \hat{A}_j$ by calculating $[\hat{A}_i, \hat{A}_j] = A_{ij}J_{ij}$ (Line 7 in Algorithm 1). The matrix J_{ij} is called Jacobi rotation [1], and it can be obtained by two steps shown in Lines 5 and 6 of Algorithm 1. *Step 1:* Compute $B_{ij} = A_{ij}^T A_{ij}$, which is called Gram matrix of A_{ij}.

Step 2: Deduce the rotation matrix J_{ij} based on EVD of B_{ij} such as $B_{ij} = J_{ij}\Lambda_{ij}J_{ij}^T$, where $J_{ij} \in \mathbb{R}^{2w\times 2w}$ is a unit orthogonal matrix, and $\Lambda_{ij} \in \mathbb{R}^{2w\times 2w}$ is a diagonal matrix whose main diagonal contains $2w$ eigenvalues of B_{ij}.

For the matrix A with n columns, there exist $\lfloor n/(2w)\rfloor$ pairs of column blocks where each column block only appears in one pair, and the corresponding $\lfloor n/(2w)\rfloor$ column block rotations can be executed concurrently at each step. $\lfloor n/w\rfloor - 1$ steps are required to complete the orthogonalization of $(\lfloor n/w\rfloor - 1) * \lfloor n/(2w)\rfloor$ pairs of column blocks with every column block pair orthogonalized exactly once, which is called a sweep. The iterative procedure converges until all the column blocks are mutually orthogonal up to working accuracy [28].

There are many methods to generate different column block index pairs (i, j) for parallel execution of $\lfloor n/(2w)\rfloor$ plane rotations in each step, including round-robin, odd-even, ring ordering, etc [12], [29], [30]. When the column block pairs are chosen systematically, the convergence rate is ultimately quadratic [1], [31].

Algorithm 1 One-sided Jacobi method using column block rotations

1: Give the width w of column blocks;
2: Rewrite A as $A = [A_1, \cdots, A_{n/w}]$ where $A_i \in \mathbb{R}^{m\times w}$;
3: **while** not converged **do**
4: **for** each pair of column blocks $A_{ij} = [A_i, A_j]$ **do**
5: $B_{ij} = A_{ij}^T A_{ij}$;
6: J_{ij} is obtained based on EVD of $B_{ij} = J_{ij}\Lambda_{ij}J_{ij}^T$;
7: $A_i = \hat{A}_i$ and $A_j = \hat{A}_j$ where $[\hat{A}_i, \hat{A}_j] = A_{ij}J_{ij}$;
8: **end for**
9: **end while**

C. One-sided Jacobi Method using Column Vector Rotations

When $w = 1$, Algorithm 1 is called the one-sided Jacobi method based on column vector rotations. For the plane rotation of the i-th and j-th column vectors $\mathbf{a}_i$ and $\mathbf{a}_j$ of A, the rotation matrix $J_{ij} \in \mathbb{R}^{2\times 2}$ can be deduced directly without using EVD of B_{ij}. The orthogonalization of $\mathbf{a}_i$ and $\mathbf{a}_j$ can be written as

$$(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_j) = (\mathbf{a}_i, \mathbf{a}_j) * J_{ij}, \text{ and } J_{ij} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix}, \quad (3)$$

where $c = \dfrac{1}{\sqrt{1+t^2}}$, $s = t * c$, and

$$t = \frac{sign(\tau)}{|\tau| + \sqrt{1+\tau^2}}, \text{ and } \tau = \frac{\mathbf{a}_i^T\mathbf{a}_i - \mathbf{a}_j^T\mathbf{a}_j}{2\mathbf{a}_i^T\mathbf{a}_j}. \quad (4)$$

Since J_{ij} can be generated directly, the one-sided Jacobi method based on column vector rotations is usually chosen in the SVD kernel design for small matrices to fully exploit the data reuse of on-chip memory in GPUs.

D. Two-sided Jacobi Method for EVD

In Algorithm 1, EVD is required to generate the Jacobi rotation matrix (Line 6). For EVD of any symmetric matrix B, the two-sided Jacobi method is to diagonalize B based on a sequence of Givens rotations $G_1, G_2, \cdots, G_k$, such as

$$G_k^T \cdots G_2^T G_1^T B G_1 G_2 \cdots G_k \to \Lambda \quad as \quad k \to \infty, \quad (5)$$

where Λ is a diagonal matrix whose main diagonal contains all the eigenvalues of B. Further, $B = J\Lambda J^T$ and $J = G_1G_2\cdots$. For each step (or elimination process), the two-sided Jacobi method selects an index pair (i, j) randomly ($i < j$), and eliminates two elements b_{ij} and b_{ji} of B by applying G_k^T and G_k on the left and right sides of B respectively, which can be viewed as a 2×2 eigenvalue problem:

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix}^T \begin{pmatrix} b_{ii} & b_{ij} \\ b_{ji} & b_{jj} \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} \hat{b}_{ii} & 0 \\ 0 & \hat{b}_{jj} \end{pmatrix}.$$

By solving the problem above, we deduce $c = \dfrac{1}{\sqrt{1+t^2}}$ and $s = t * c$, where

$$t = \frac{sign(\rho)}{|\rho| + \sqrt{1+\rho^2}}, \text{ and } \rho = \frac{b_{ii} - b_{jj}}{2b_{ij}}.$$

It should be noted that the rows and columns in B are updated at the same time. Hence, the two-sided Jacobi method has to be executed sequentially [32].

III. W-CYCLE SVD

A. Observation

This work is motivated by two attractive observations.

Observation 1. In Algorithm 1, by denoting $A_{ij} = U_{ij}\Sigma_{ij}V_{ij}^T$ as the SVD of A_{ij}, we observe that the right singular matrix V_{ij} of A_{ij} equals to the eigenvector matrix J_{ij} of B_{ij}, which means that SVD of A_{ij} is equivalent to EVD of B_{ij} for the generation of the rotation matrix J_{ij}. This observation implies that we can avoid GEMM $B_{ij} = A_{ij}^T A_{ij}$ in Line 5 of Algorithm 1 by directly performing SVD of A_{ij} to generate J_{ij} instead of executing EVD of B_{ij}. Therefore, the SVD process of A can be regarded recursively as multiple SVDs of sub-matrices A_{ij}. As Figure 1 shows, the one-sided Jacobi method based on SVD of A_{ij} within GPU shared memory (SM) is faster than using EVD of B_{ij} within SM. However, SVD of A_{ij} in GPU global memory (GM) is slower than EVD of B_{ij} in SM (Figure 1). Thus, in our design, if A_{ij} can be stored entirely in SM, we leverage the SVD of A_{ij} to obtain J_{ij} directly. Otherwise, we further consider whether the EVD of B_{ij} can be executed in SM.

Observation 2. The selection of w affects the performance of the batched SVD in the following two aspects.

First, w affects both the usage of SM and the convergence speed of the one-sided Jacobi method. On the one hand, as Figure 2 shows, the number of rotations per sweep decreases with w increasing, which leads to a faster convergence speed. On the other hand, w has to be small enough to ensure that A_{ij} or B_{ij} can be stored entirely in SM for efficiently generating J_{ij}. Figure 2 also shows that, when $w > 24$, both SVD of A_{ij} and EVD of B_{ij} can not be entirely implemented within SM, which results in longer execution time.

Second, it is not always optimal to select a uniform w for different matrices in a batched SVD. For instance, we consider the SVDs of two matrices A^1 and A^2 of size 32×1024 and 1024×1024 respectively on NVIDIA V100 GPU, which

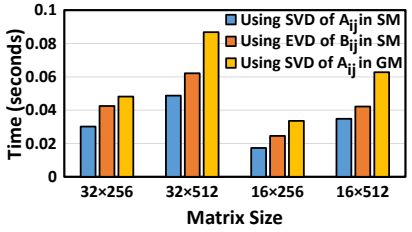


Fig. 1: Time of one-sided Jacobi methods in different cases.

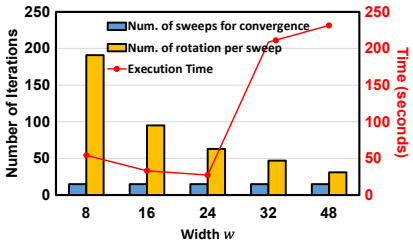


Fig. 2: One-sided Jacobi method for a batched SVD of 100 matrices with each size of 1536×1536 on NVIDIA V100 GPU.

provides the static SM capacity of 48KB per thread block. To ensure that either SVD of A_{ij}^k or EVD of B_{ij}^k can be executed entirely within SM ($k = 1, 2$), a uniform w must be no larger than 24. However, if we set different w for A^1 and A^2, i.e., w_1 and w_2, there is only a constraint for $w_2 \leq 24$. For the SVD of A^1, we can set for example $w_1 = 48$, which ensures the SVD of A_{ij}^1 implemented within SM and achieves better convergence.

B. Multilevel Perspective

Inspired by the multi-grid method [33], we perform the column block rotations of different matrices in a batched SVD at different dedicated levels determined by different w, which is the key idea underlying the W-cycle SVD. The design of W-cycle SVD is based on the following theoretical result, which provides a recursive view of the one-sided Jacobi method from a multilevel perspective.

Theorem 1: Assume that A is a $m \times n$ matrix and $B = A^T A$. A unit orthogonal matrix $V \in \mathbb{R}^{n \times n}$ makes sure that $B = V \Lambda V^T$ is the EVD of B where $\Lambda \in \mathbb{R}^{n \times n}$ is a diagonal matrix and all the eigenvalues of B are arranged along the main diagonal of Λ, if and only if there exists another unit orthogonal matrix $U \in \mathbb{R}^{m \times m}$ such that $A = U \Sigma V^T$ is a SVD of A where $\Sigma \in \mathbb{R}^{m \times n}$ places all singular values of A along its main diagonal and its off-diagonal elements vanish.

Remark. Theorem 1 reveals that SVD of A_{ij} is equivalent to EVD of B_{ij} for obtaining the rotation matrix J_{ij}.

Assume that there are μ matrices $A^1, \cdots, A^\mu$, and the size of A^k is $m_k \times n_k$ ($k = 1, 2, \cdots, \mu$). Without loss of generality, we consider the SVD of A^k, which is placed at the highest level, i.e., Level 0. Next, A^k is divided into multiple column blocks A_i^k with the width w, leading to the sub-matrices $A_{ij}^k = [A_i^k, A_j^k]$, which are placed to Level 1. For convenience, we rewrite A_{ij}^k as $A_{ij}^{(1,k)}$, and denote sub-matrices generated from

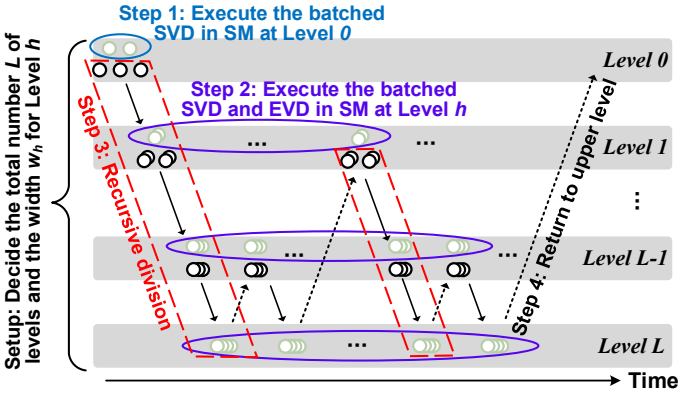


Fig. 3: Overview of W-cycle SVD. The green circles at each level represent the matrices belonging to the first or second group, and the batched SVD or EVD in SM is applied to them. The black circles represent the matrices in the third group, in which each matrix is further divided into smaller sub-matrices at the next level. The sketch of the multilevel workflow is like the capital ''W''.

A^k at Level h as $A_{ij}^{(h,k)}$ hereafter. According to Theorem 1, the column block rotation of A_i^k and A_j^k needs the SVD of $A_{ij}^{(1,k)}$ to generate the rotation matrix $J_{ij}^{(1,k)}$. In other words, the SVD of A^k is disassembled into SVDs of $A_{ij}^{(1,k)}$ at Level 1. Further, we can recursively summarize a multilevel workflow: The SVD of $A_{ij}^{(h,k)}$ with the column block width w_h at Level h involves the SVDs of $A_{ij}^{(h+1,k)}$ with the smaller width w_{h+1} at Level $h+1$. The workflow continuously goes down to the SVDs of the smallest sub-matrices $A_{ij}^{(L,k)}$ at Level L. If either $A_{ij}^{(L,k)}$ or $B_{ij}^{(L,k)} = (A_{ij}^{(L,k)})^T A_{ij}^{(L,k)}$ is small enough, the SVD of $A_{ij}^{(L,k)}$ or EVD of $B_{ij}^{(L,k)}$ would be executed within SM to obtain the rotation matrix $J_{ij}^{(L,k)}$ at Level L. The workflow returns to the upper level until the SVD on the current level is completed. Since each SVD needs a sequence of rotations to ensure that all the column blocks are orthogonalized with each other, SVD at Level h would call SVD at Level $h+1$ repeatedly, which makes the sketch of workflow like the capital ''W'', as shown in Figure 3.

C. Multilevel Algorithm

Based on the analysis above, we propose W-cycle SVD, a new multilevel algorithm for batched SVD on GPUs (Algorithm 2). The multilevel workflow of W-cycle SVD is shown in Figure 3.

Setup: Decide the total number L of levels and the width w_h for Level h. Based on Observation 2, a large w_1 is selected. For $h \geq 2$, w_h is determined by a given selection way, which ensures that $w_{h+1} < w_h$, and EVD of any $2w_L \times 2w_L$ matrix can be implemented entirely in SM at Level L. In this work, an auto-tuning engine is proposed to select w_h (Section IV-D3).

Step 1: Execute the batched SVD in SM at Level 0. All the matrices A^k are at Level 0 ($k = 1, \cdots, \mu_h$). Based on Observation 1, a batched SVD kernel is applied to the matrices A^k whose SVDs can be entirely executed within SM (Line 3

Algorithm 2 W-cycle SVD for Batched SVD

1: Input a batch of matrices $A^1, \cdots, A^\mu$, and the size of A^k is $m_k \times n_k$ $(k = 1, 2, \cdots, \mu)$; a given width w for column blocks;
2: **if** SVD of A^k can be implemented entirely within SM **then**
3: Compute SVD of A^k by using batched SVD kernel in SM;
4: **else**
5: Rewrite A^k as $A^k = [A^k_1, \cdots, A^k_{n_k/w}]$ where $A^k_i \in \mathbb{R}^{m_k \times w}$;
6: **while** not converged **do**
7: **for** each pair of column blocks $A^k_{ij} = [A^k_i, A^k_j]$ **do**
8: **if** SVD of A^k_{ij} can be accomplished entirely within SM **then**
9: Compute J^k_{ij} based on SVD of $A^k_{ij} = U^k_{ij}\Sigma^k_{ij}(J^k_{ij})^T$ by using batched SVD kernel in SM;
10: **else if** EVD of $B^k_{ij} = (A^k_{ij})^T A^k_{ij}$ can fit in SM **then**
11: Compute J^k_{ij} based on EVD of $B^k_{ij} = J^k_{ij}\Sigma^k_{ij}(J^k_{ij})^T$ by using batched EVD kernel in SM;
12: **else**
13: Update w by following a given selection way;
14: Use W-cycle SVD with w for A^k_{ij} to obtain J^k_{ij};
15: **end if**
16: $A^k_i = \hat{A}^k_i$ and $A^k_j = \hat{A}^k_j$ where $[\hat{A}^k_i, \hat{A}^k_j] = A^k_{ij}J^k_{ij}$;
17: **end for**
18: **end while**
19: **end if**

in Algorithm 2). The rest of the matrices go down from Level 0 to Level 1.

Step 2: Execute the batched SVD and EVD in SM at Level h. When a matrix $A^{(h-1,k)}$ goes from Level $h-1$ to Level h (where $A^{(0,k)} = A^k$), $A^{(h-1,k)}$ is partitioned to several column blocks with the width w_h. The resulting multiple $A^{(h,k)}_{ij}$ at Level h are divided into three groups: (i) all the sub-matrices $A^{(h,k)}_{ij}$ whose SVD can be accomplished in SM; (ii) the sub-matrices $A^{(h,k)}_{ij}$ satisfying that SVD of $A^{(h,k)}_{ij}$ can not be implemented in SM, but EVD of $B^{(h,k)}_{ij} = (A^{(h,k)}_{ij})^T A^{(h,k)}_{ij}$ can be done in SM; (iii) the rest of sub-matrices. For the first two groups, batched SVD and EVD kernels are used respectively to obtain the rotation matrices (Line 9 and Line 11 in Algorithm 2). The third group goes to Level $h+1$ (Line 14 in Algorithm 2).

Step 3: Recursive division. Step 2 is repeated recursively for the third group until the workflow arrives at Level L.

Step 4: Return to upper level. After the column block rotations of sub-matrices in the third group are completed, the workflow goes backward to the upper level. When the workflow returns Level 0 again, a W-cycle sweep is ended. If the SVD of a matrix is accomplished at any sweep, it exits the workflow.

After the workflow converges, the right singular matrices (i.e., V) for all the SVDs would be obtained. Finally, the left singular matrices (i.e., U) and singular value matrices (i.e., Σ) can be easily deduced.

Remark. The multilevel design naturally avoids using a uniform w for all the matrices in batched SVD. Specifically, the W-cycle SVD keeps different column block rotations at different levels, which brings two advantages. (1) the generations of all the rotation matrices at any level are implemented within SM, which fully exploits the data reuse of SM. (2) w is selected properly for each matrix at a dedicated level, which ensures the optimal convergence speed of batched SVD.

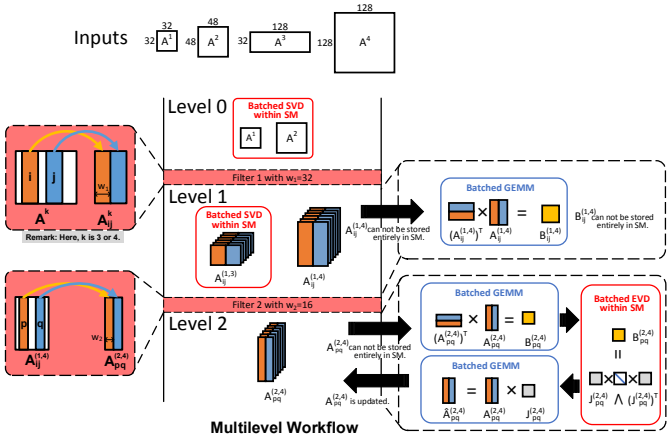


Fig. 4: An example of W-cycle SVD for four matrices.

D. A Brief Example

Figure 4 illustrates an example of the W-cycle SVD for four matrices. Initially, four matrices A^k (where $k = 1, \cdots, 4$) are at Level 0. The batched SVD kernel performs the SVDs of A^1 and A^2 since both of them can be entirely implemented in SM. Next, a large w is selected ($w_1 = 32$) for A^3 and A^4 to generate $A^{(1,3)}_{ij}$ and $A^{(1,4)}_{ij}$ which are placed at Level 1. As the SVD of $A^{(1,3)}_{i,j}$ can be entirely implemented within SM, the rotation matrix $J^{(1,3)}_{ij}$ is generated at Level 1. Meanwhile, $A^{(1,4)}_{i,j}$ is further divided into $A^{(2,4)}_{p,q}$ which is processed recursively in Level 2 with shrinking w ($w_2 = 16$). Since $A^{(2,4)}_{pq}$ is too large for SM while EVD of $B^{(2,4)}_{pq}$ fits into SM, the batched EVD of $B^{(2,4)}_{pq}$ is conducted at Level 2 to obtain the rotation matrix $J^{(2,4)}_{pq}$. After the index pairs (p, q) traverse all of the possible choices, the workflow goes back to Level 1. At Level 1, the index pairs (i, j) are changed. Until all the possible (i, j) are chosen exactly once, the workflow returns to Level 0. The process above repeats for A^3 and A^4. If all the column blocks of A^3 and A^4 are orthogonal with each other, the SVD is completed.

Remark. It is clear that different w are applied as multiple filters. When $A^{(h-1,k)}$ goes from Level $h-1$ to Level h, it has to pass through a filter. Meanwhile, $A^{(h-1,k)}$ is divided to several column blocks, and each pair (i, j) of them forms $A^{(h,k)}_{ij}$ placed at Level h. During all the matrices go through multiple filters with the decease of w, W-cycle SVD finds a dedicated w as large as possible for each input matrix.

Although the specific w for each input matrix also can be statically determined before executing batched SVD, the proposed recursion based on multilevel workflow has three advantages.

First, the recursion establishes a hierarchical organization mode to pair column blocks for rotations, which ensures both high convergence speed and data locality. However, the static way determining w without recursion is equivalent to directly executing rotations at the last level, which ignores the information at higher levels for organizing column block pairs.

Second, the static way involves multiple individual workflows for SVDs with different w, which is still size-sensitive.

However, the recursion achieves a uniform workflow for all the SVDs, which is size-oblivious and takes the performance improvement chances from our two observations.

Third, the recursion adaptively places each rotation at the right level without any preprocessing overhead.

IV. Implementation and Optimization

In the W-cycle SVD, the multilevel workflow requires batched SVD and EVD kernels in SM, as shown in Figure 4. Moreover, the Gram matrix computation and the column block update involve two batched GEMMs at each level. In this section, we present the efficient kernel design of the W-cycle SVD.

A. Challenges

Challenge 1: How to design efficient batched SVD and EVD kernels using SM for sub-matrices at each level. The recent work has shown that the thread-level parallelism of SVD and EVD kernels within SM is hard to be improved [19]. First, the batched SVD kernel design usually exploits the data reuse of SM by using the one-sided Jacobi method based on column vector rotations which generates the rotation matrix directly. However, calculating the parameters c and s of J_{ij} involves three inner products of vectors in Equation (4), which is not friendly to the thread-level parallelism due to the decrease of active threads during the summation process. Hence, W-cycle SVD requires an efficient task assignment mechanism to improve the parallelism of column vector rotations. Meanwhile, the batched EVD kernel within SM usually needs the two-sided Jacobi method, which applies the rotation matrix and its transpose on the right and left sides of the decomposed matrix simultaneously and updates rows and columns at the same time (Section II-D). Therefore, when the i-th and j-th of both rows and columns are updated by J_{ij}, the rest elements in other rows and columns can not be changed, which leads to a sequential implementation and limits the potential parallelism. To break this bottleneck, W-cycle SVD needs a new EVD kernel that could update all the elements of the decomposed matrix in parallel.

TABLE I: Different tile sizes for two batched GEMMs at Level 1 of W-cycle SVD with two levels for 100 matrices.

Matrix Size	Height of Tile	32	64	128	256	512
	Width of Tile	Time of Batched SVD (seconds)				
256×256	8	0.44	0.40	0.39	0.40	-
	16	0.21	0.20	0.20	0.18	-
	32	0.15	0.14	**0.13**	0.15	-
	48	0.19	0.20	0.18	0.20	-
512×512	8	3.47	3.14	3.04	3.05	3.00
	16	1.64	1.49	1.44	1.44	1.53
	32	0.99	0.99	0.95	**0.94**	0.95
	48	1.11	1.04	1.01	1.00	1.00

* For the last column in different cases, the tile height equals to the row number, which means that each GEMM is assigned to a thread block.

Challenge 2: How to design efficient batched GEMM kernels at each level. For batched GEMM, it is intuitive to assign each GEMM task to one thread block. However, it introduces two implementation issues. First, in the case of small batch size and small matrix column numbers, the task assignment above leads to few active thread blocks and low thread-level parallelism. Second, different matrices may involve different row numbers, which results in unbalanced workloads for thread blocks. To solve these two issues, we assign each GEMM task to multiple thread blocks, rather than only one block as usual. However, the different tile sizes lead to different data reuse rates of GEMMs and thus affect the performance of batched SVD as shown in Table I. Therefore, W-cycle SVD requires an adaptive approach for the assignment of batched GEMM tasks.

B. Batched SVD Kernel Design Based on Shared Memory

The batched SVD kernel within SM is designed based on column vector rotations (Section II-C). To maximize the data reuse, we assign a matrix to one thread block, and keep it in SM until its SVD is completed. For a $m \times n$ matrix A, if $m < n$, we execute the SVD of A^T instead of A as the SVD process of A^T involves fewer iterative steps. When the SVD of A^T is completed, i.e., $A^T = \hat{U}\hat{\Sigma}\hat{V}^T$, the SVD of A can be obtained by $A = \hat{V}\hat{\Sigma}\hat{U}^T$. Without loss of generality, we assume that $n \leq m$ in the following discussion. The batched SVD kernel design is to solve two issues: (1) how to assign the column-orthogonalization tasks to the threads; (2) how to avoid the vector inner products in Equation (4) for calculating the parameters c and s of J_{ij}.

1) Assignment of Orthogonalization Tasks: To fully exploit the parallelism of the one-sided Jacobi method, we assign an orthogonalization task of a pair of columns to α thread warp rather than one thread [4], [19], where α is chosen from a set $\{1, 1/2, 1/4, 1/8\}$. To determine the value of α, we propose two methods. Considering the fact one warp usually includes 32 threads, the first method is to calculate the greatest common factor β of m^* and 32 where $m^* = \max\{m_k\}$, and set $\alpha = \max\{4, \beta\}/32$. For example, if $m^* = 48$, we have $\beta = 16$ and $\alpha = 1/2$, which means assigning 16 threads for a pair of columns. The second one is a machine learning method for training a decision tree to determine α. We choose m^* (the largest row number) and μ (the batch size) as the features, and the optimal α as the label. The data set is collected by randomly generating thousands of batched GEMMs and determining the right label for each batch based on practical tests. For the decision tree, each node makes choice by performing a comparison. For example, the root node compares m^* with a value within this node. If m^* is larger than this value, the tree goes deeper along the right branch. The second node compares μ with a value, and then it arrives at a leaf node according to the comparison result. The leaf node is a vector with 4 elements which correspond to the probabilities to choose the four candidate values of α. The two methods above both can improve the thread-level parallelism for the batched SVD in SM.

2) Optimization of Inner Products: To orthogonalize two column vectors $\mathbf{a}_i$ and $\mathbf{a}_j$, Equation (4) involves three inner products, i.e., $\mathbf{a}_i^T\mathbf{a}_i$, $\mathbf{a}_i^T\mathbf{a}_j$ and $\mathbf{a}_j^T\mathbf{a}_j$. Our optimization aims

to avoid the execution of inner products as much as possible. According to Equation (3), we can derive

$$\begin{cases} \hat{\mathbf{a}}_i^T \hat{\mathbf{a}}_i = c^2 \cdot \mathbf{a}_i^T \mathbf{a}_i + 2cs \cdot \mathbf{a}_i^T \mathbf{a}_j + s^2 \cdot \mathbf{a}_j^T \mathbf{a}_j, \\ \hat{\mathbf{a}}_j^T \hat{\mathbf{a}}_j = s^2 \cdot \mathbf{a}_i^T \mathbf{a}_i - 2cs \cdot \mathbf{a}_i^T \mathbf{a}_j + c^2 \cdot \mathbf{a}_j^T \mathbf{a}_j. \end{cases} \tag{6}$$

In the one-sided Jacobi method based on column vector rotations, assume that the values of $\mathbf{a}_i^T \mathbf{a}_i$ and $\mathbf{a}_j^T \mathbf{a}_j$ are recorded at the previous orthogonalization rotation. According to Equation (4), we only need the inner product $\mathbf{a}_i^T \mathbf{a}_j$ to deduce c and s for the current orthogonalization rotation. Meanwhile, by Equation (6), $\hat{\mathbf{a}}_i^T \hat{\mathbf{a}}_i$ and $\hat{\mathbf{a}}_j^T \hat{\mathbf{a}}_j$ can be obtained directly and recorded for the next rotation. The approach above successfully avoids two-thirds of inner product operations in the orthogonalization process.

C. Batched EVD Kernel Design Based on Shared Memory

In W-cycle SVD, since any matrix $A_{ij}^{(h,k)}$ at Level h has the same column width of $2w_h$, $B_{ij}^{(h,k)} = (A_{ij}^{(h,k)})^T A_{ij}^{(h,k)}$ is a $2w_h \times 2w_h$ square matrix. According to the analysis in Challenge 1, when the two-sided Jacobi method updates the i-th and j-th of both rows and columns, the rest elements can not be changed. Hence, for the EVD of a 20×20 matrix, there are at most 80 active concurrent threads with each thread updating one element in two rows and two columns ($(2+2) \times 20 = 80$ elements), which limits the parallelism. To address this issue, we design a new EVD kernel updating all the elements of the decomposed matrix in parallel.

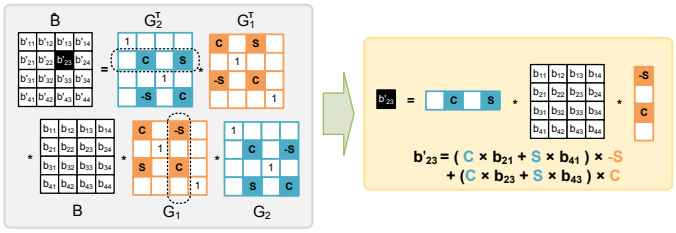


Fig. 5: Parallelization of the two-sided Jacobi method.

If the elimination pairs are chosen randomly or following the common predetermined order such as $(1,2)$, $(1,3)$, $\cdots$, $(1,2w_h)$; $(2,3)$, $(2,4)$, $\cdots$, $(2,2w_h)$; $\cdots$; $(2w_h-1, 2w_h)$, it is not possible to execute different elimination processes concurrently. For example, if a row vector is involved in two adjacent elimination processes, the second process can not begin until the first one is ended. In order to achieve parallel implementation, it is necessary to generate a sequence of elimination pairs such that any index in $\{1, 2, \cdots, 2w_h\}$ appears in a pair only once. Inspired by the one-sided Jacobi method, we choose the round-robin approach [12], [29] to determine these pairs. As there are $w_h(2w_h-1)$ possible pairs, all the pairs are generated in $2w_h - 1$ steps with each step processing w_h eliminations in parallel.

For each elimination pair (i,j), we construct the corresponding Givens matrix G_{ij}, and write each step as

$$\hat{B} = G_{pq}^T \cdots G_{ij}^T B G_{ij} \cdots G_{pq}, \tag{7}$$

where $(i,j), \cdots, (p,q)$ constitute a round-robin sequence. As any index from $\{1, 2, \cdots, 2w_h\}$ appears only in one pair, the transposed Givens matrices update different rows of B in parallel, and the Givens matrices change different columns of B concurrently. This fact indicates that any element of $\hat{B}$ in Equation (7) can be calculated by $x^T B y$, where y is a column vector chosen from a Givens matrix and x^T is a row vector chosen from the corresponding transposed matrix. Figure 5 shows an example. The element b'_{23} in $\hat{B}$ involves the second row x^T of G_2^T and the third column y of G_1. Thus, we have $b'_{23} = x^T B y$. As any row or column of Givens matrix has at most two non-zero elements, calculating each element of $\hat{B}$ only needs 6 multiplications and 3 additions (or subtractions) as shown in Figure 5. Consequentially, all the elements of $\hat{B}$ can be calculated in parallel.

D. Tailoring Strategy

In the following, we consider the optimization of the two batched GEMMs at each level of W-cycle SVD.

Assume that there are μ_h matrices $A_{ij}^{(h,k)}$ at Level h, and their Gram matrices are denoted as $B_{ij}^{(h,k)}$. At Level h, the first batched GEMM is $B_{ij}^{(h,k)} = (A_{ij}^{(h,k)})^T A_{ij}^{(h,k)}$. After obtaining the rotation matrix $J_{ij}^{(h,k)}$, the second batched GEMM is to update $A_{ij}^{(h,k)}$ as $\hat{A}_{ij}^{(h,k)} = A_{ij}^{(h,k)} J_{ij}^{(h,k)}$. As $i, j = 1, \cdots n_k/w_h$ and $k = 1, \cdots \mu_h$, the batch size of both batched GEMMs is $\sum_{k=1}^{\mu_h} \frac{n_k}{2w_h}$. According to the analysis in Challenge 2, the different sizes of $A_{ij}^{(h,k)}$ impact the thread-level parallelism and data reuse rate of batched GEMM. Hence, our design aims to improve the thread-level parallelism and data reuse rate in the two batched GEMMs.

1) Tailoring Design: As increasing the number of current thread blocks could improve the thread-level parallelism, our basic idea is to tailor $A_{ij}^{(h,k)}$ into multiple segments along the row direction (Figure 6(b)). By this way, a GEMM task can be assigned to multiple thread blocks. For the first batched GEMM, each segment is multiplied by its transposition from the left side in one thread block. Then, the corresponding results from multiple thread blocks are summed up to obtain $B_{ij}^{(h,k)}$ (Figure 6(c)). For the second batched GEMM, each thread block concurrently updates each segment by multiplying the segment with the corresponding rotation matrix $J_{ij}^{(h,k)}$ (Figure 6(d)).

In our tailoring strategy, the task assignment involves three steps. Firstly, we build a standard plate (SP) of size $\delta_h \times 2w_h$. Next, we tailor each $A_{ij}^{(h,k)}$ into multiple standard segments with SP size along the row direction. The residual rows are viewed as a segment of size $(m_k - \lfloor m_k/\delta_h \rfloor \cdot \delta_h) \times 2w_h$. Thirdly, each standard segment is assigned to a thread block. The residual segments are assigned to a thread block until the sum of their row numbers exceeds an empirical parameter $1.2\delta_h$, then a new thread block is used, and so on.

So far, the values of δ_h and w_h, which are essential for the performance of batched GEMM, remain undetermined. To find the optimal values of them, we model the effects of the tailoring strategy on the thread-level parallelism and data

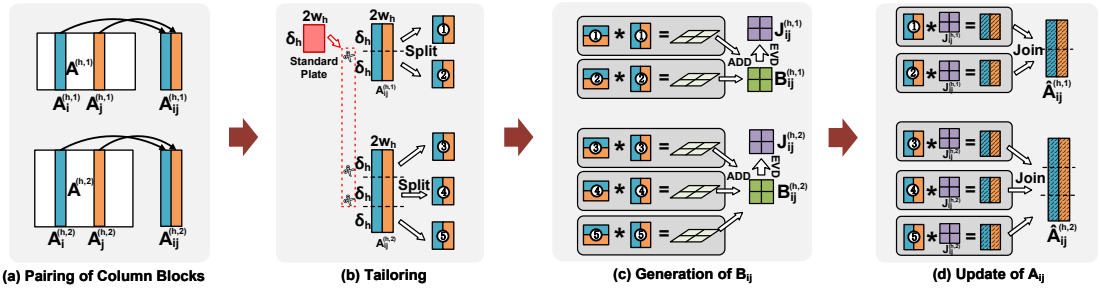


Fig. 6: Tailoring strategy.

reuse rate respectively. Based on the performance models, we construct an optimization problem to determine δ_h and w_h.

2) Performance Models: We define the thread-level parallelism (TLP) as the number of threads for a batched GEMM. With a $\delta_h \times 2w_h$ SP and the number of threads T_h per thread block, TLP_1 and TLP_2 for the two batched GEMMs described above can be calculated as follows:

$$TLP_1 = TLP_2 \approx \sum_{k=1}^{\mu_h} \frac{n_k \cdot m_k}{2w_h \cdot \delta_h} T_h. \tag{8}$$

It is clear that both TLP_1 and TLP_2 decrease as the SP size $\delta_h \times 2w_h$ increases. Furthermore, the number of threads T_h can also affect parallelism. When T_h increases, there are more threads to exploit TLP.

To model the data reuse rate, we choose the arithmetic intensity (AI) [34] as a quantitative index. AI is the number of arithmetic instructions per byte of the memory request. Larger AI means a higher data reuse rate. The numbers of load instructions for each thread in two batched GEMMs are

$$\begin{cases} num\_load_1 \approx \frac{2w_h \cdot \delta_h}{Load\_width \cdot T_h}, \\ num\_load_2 \approx \frac{2w_h \cdot \delta_h + 2w_h \cdot 2w_h}{Load\_width \cdot T_h}, \end{cases}$$

where $Load\_width$ is the number of data in one load request. Moreover, the number of arithmetic FMA instructions for each thread can be approximated as: $num\_FMA_1 = num\_FMA_2 \approx \frac{2w_h \cdot 2w_h \cdot \delta_h}{T_h}$. Further, the ratio of arithmetic instruction to load instruction, i.e., num_FMA/num_Load, leads to AI_1 and AI_2 for the two batched GEMMs:

$$\begin{cases} AI_1 = \frac{num\_FMA_1}{num\_load_1} \approx Load\_width \cdot 2w_h, \\ AI_2 = \frac{num\_FMA_2}{num\_load_2} \approx Load\_width \cdot \frac{2w_h \cdot \delta_h}{2w_h + \delta_h}. \end{cases} \tag{9}$$

3) Auto-tuning Engine: To determine the optimal δ_h and w_h, we build a multiple-objective programming problem based on the performance models above. Empirically, we give higher priority to TLP and then optimize AI. Hence, the first objective is to maximize $TLP_1 + TLP_2$. Although smaller w_h leads to larger TLP, it also produces more column blocks, resulting in more orthogonalization processes in a sweep. In order to limit the number of orthogonalization processes, maximizing AI_1 is chosen as the second objective due to the linear relationship of AI_1 and w_h. Then, maximizing AI_2 is defined as the third objective. Combining the three objectives above, the multiple-objective programming problem can be described as

$$\max_{\delta_h, w_h, T_h} \begin{cases} f_1 = \sum_{k=1}^{\mu_h} \frac{n_k \cdot m_k}{w_h \cdot \delta_h} T_h, \\ f_2 = Load\_width \cdot 2w_h, \\ f_3 = Load\_width \cdot \frac{2w_h \cdot \delta_h}{2w_h + \delta_h}. \end{cases} \tag{10}$$

To solve the optimization problem (10), we propose an efficient method with two steps.

TABLE II: Tailoring parameters.

No.	w_h	δ_h	T_h
1	48	m^*	256
2	24	m^*	256
3	24	$m^*/2$	256
4	16	$m^*/2$	256
5	16	$m^*/4$	256
6	16	$m^*/8$	256
7	8	$m^*/4$	128
8	8	$m^*/8$	128

TABLE III: Available plans.

No.	w_h	δ_h	T_h
1	48	256	256
2	24	256	256
3	24	128	256
4	16	128	256
5	16	64	256
6	16	32	256
7	8	64	128
8	8	32	128

The first step is to generate a series of candidate solutions (the effective tailoring parameters) as shown in Table II. First of all, we give higher priority for $T_h = 256$ than $T_h = 128$, since larger T_h leads to higher TLP. Next, to make sure that either SVD of $A_{ij}^{(h,k)}$ or EVD of $B_{ij}^{(h,k)}$ can be implemented entirely in SM, we set $w_h \leq 48$ (Observation 2 in Section III-A) and select δ_h sequentially from a set $\{m^*, m^*/2, m^*/4, m^*/8, \cdots\}$ where m^* is the largest row number of all the matrices. Further, we arrange all the candidate solutions in the increasing order of TLP and the descending order of AI (Table II), which essentially determines the search direction. Specifically, with the solution index increasing, f_1 increases while f_2 and f_3 decrease.

The second step is to search for the optimal solution (the optimal tailoring parameters). At this step, we pick out candidate solutions sequentially from Table II until the selected solution makes f_1 larger than a predefined threshold. For a given platform, we determine the threshold by evaluating all the tailoring solutions in Table II for two batched GEMMs in SVD of a huge matrix, and calculating the different overall TLPs of different solutions. We choose the threshold as the TLP which leads to the inflection point with no significant performance improvement anymore. The threshold is determined only once for a particular platform.

Specifically, we present an example to show how the method above works. Assume that there are 100 matrices with the same size of 256×256. Firstly, we generate all the available tailoring plans in Table III with $m^* = 256$. The threshold is determined as $306,149$ on NVIDIA Tesla V100 GPU. Next, we select the first candidate from Table III, and derive $f_1 = 68,267$, which is smaller than the threshold. Then, the next solution is checked, and so on. The check process ends until the fourth solution is selected, which provides the final tailoring parameter with $f_1 = 409,600$.

V. Evaluation

As an illustration of the effectiveness of the proposed W-cycle SVD, this section presents the performance evaluation of W-cycle SVD alone and W-cycle SVD combined with the tailoring strategy. Most of the evaluation results are tested on NVIDIA Tesla V100 GPU. To evaluate the portability of W-cycle SVD, we further perform the evaluation on different GPU architectures, such as NVIDIA Ampere A100, Tesla P100, GTX Titan X, and AMD Vega20 GPUs. The TLP threshold is set as $306,149$ for the auto-tuning engine. In addition, we use the C language for programming with CUDA-10.1 and ROCm-3.5.

In the evaluation, the cuSOLVER [20] is used as the baseline, which provides the batched SVD API requiring the matrix size m and n smaller than 32. For larger matrices not supported by the batched SVD API, the baseline is set to serially call a single SVD API in cuSOLVER. Further, we also show the comparison between W-cycle SVD and MAGMA [22], [35] on GPUs. Finally, we demonstrate that the W-cycle SVD achieves performance speedup in a real-world application: the data assimilation in oceanic models [36]–[38].

A. Evaluation of W-cycle SVD

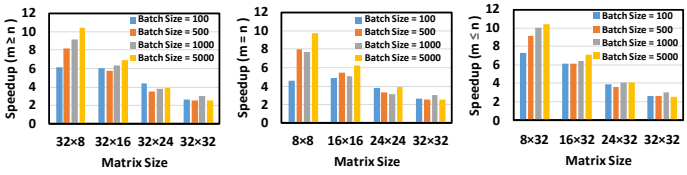


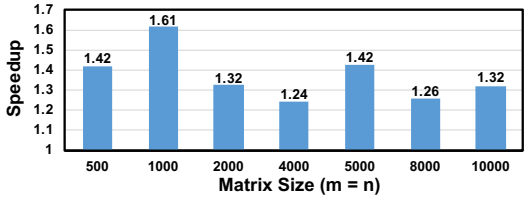
Fig. 7: W-cycle SVD for improvement over cuSOLVER.

1) Performance Comparison with Batched SVD Kernel Function in cuSOLVER: Figure 7 shows the performance comparison of W-cycle SVD with the batched SVD kernel in cuSOLVER. In the experiment, two batched GEMMs at each level of W-cycle SVD are implemented using the common way with a thread block for one GEMM. Since the numbers of rows and columns are no larger than 32, each SVD can be executed entirely within SM. It is clear that W-cycle SVD achieves $2.6 \sim 10.2\times$ speedup. Three observations from the results are presented as follows.

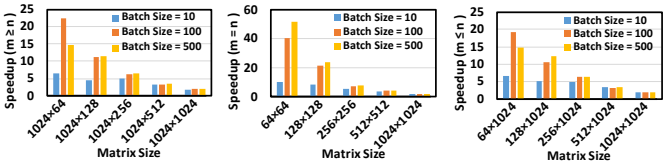
Firstly, for the fixed m and n, the performance benefit of W-cycle SVD increases with the growth of the batch size, which owes to our batched SVD kernel design based on SM. When the batch size is small, using α warp for each orthogonalization task provides a higher degree of potential parallelism. When the batch size is large, avoiding two-thirds of vector inner products improves the performance significantly.

Secondly, for a given batch size, the performance benefit of W-cycle SVD decreases with the increase of matrix size. For instance, when the batch size is 100, the speedup achieves $7.2\times$ for the 8×32 matrices, while the speedup becomes $2.7\times$ for the 32×32 matrices. The reason is that our batched SVD kernel leads to higher parallelism for SVDs of smaller matrices.

Thirdly, compared with the batched SVD with $m > n$, W-cycle SVD achieves higher speedup for SVDs with $m \leq n$, because SVDs of the transpose of the matrices with $m \leq n$ are executed to decrease the iteration number for each sweep.



(a) Comparison with cuSOLVER using batch size=1.



(b) Comparison with cuSOLVER using various batch sizes.

Fig. 8: W-cycle SVD for performance improvement.

2) Performance Comparison with Batched SVD Implementation Based on cuSOLVER: Figure 8 shows the performance comparison between W-cycle SVD and a batched SVD implementation using SVD kernel in cuSOLVER for matrices of size larger than 32×32. As Figure 8(a) shows, when the batch size is one, W-cycle SVD achieves $1.37\times$ speedup on average compared with cuSOLVER, which indicates that W-cycle SVD also has high performance for single SVD. This is because our batched EVD kernel realizes the parallel update.

Figure 8(b) gives the test results with various batch sizes. W-cycle SVD achieves $2 \sim 20\times$ performance speedup over cuSOLVER. We highlight that the benefit of W-cycle SVD is consistent as the batch size increases, which implies that our design has significant performance improvement. This mainly owes to the multilevel design achieving both high data reuse and fast convergence speed simultaneously.

3) Performance Comparison with Batched SVD Implementation Based on MAGMA: Figure 9 shows the performance comparison between W-cycle SVD and a batched SVD implementation using SVD kernel in MAGMA. For single SVD, W-cycle SVD achieves at least $2.78\times$ speedup compared with MAGMA. For batched SVD, the speedup is always larger than $4.2\times$ in different cases, and the benefit of W-cycle SVD is consistent with the increase of the batch size.

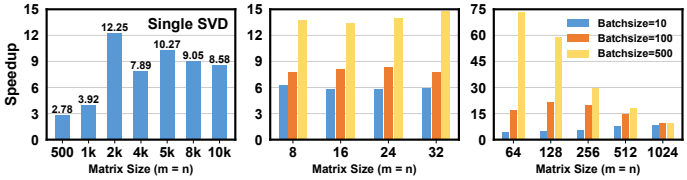


Fig. 9: W-cycle SVD for improvement over MAGMA.

TABLE IV: Time (seconds) for SVDs of 200 Matrices with the same size on P100 GPU.

Matrix Size ($m = n$)	100	128	256	512
Batched DP Direct [19]	0.103	0.211	1.395	10.41
Batched DP Gram [19]	0.132	0.253	1.294	7.316
cuSOLVER	0.527	1.134	2.549	9.750
W-cycle SVD	0.012	0.051	0.316	2.012

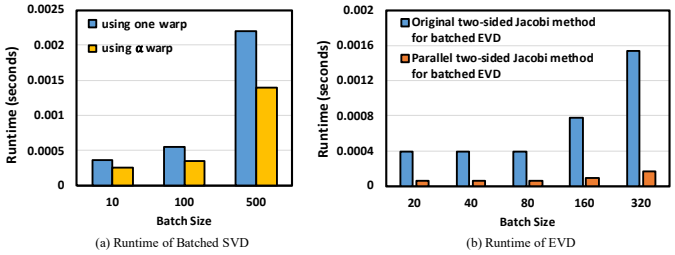
4) Performance comparison with the state-of-the-art batched SVD methods: Based on NVIDIA Tesla P100 GPU, several optimized batched SVD implementations are presented in [19], and the reported performance improvement over cuSOLVER is impressive. We compare W-cycle SVD with the state-of-the-art batched SVD methods in [19] by collecting the results of our algorithm with the same experimental hardware and software parameters, e.g., batch size and matrix size.

As shown in Table IV, both W-cycle SVD and implementations in [19] are faster than cuSOLVER. We highlight that W-cycle SVD further achieves $4.1 \sim 8.6\times$ and $3.6 \sim 11\times$ speedups respectively compared with Batched_DP_Direct, Batched_DP_Gram in [19].

B. Analysis on Speedups

Compared with cuSOVLER and MAGMA, W-cycle SVD achieves at least $1.24\times$ performance speedup for the single SVD, and at least $2.2\times$ performance speedup for the batched SVD. Our analysis on the speedup focuses on parallelism and locality through benchmarks and GPU profiling data.

Parallelism. In W-cycle SVD, the batched SVD kernel uses α warp for the column rotation tasks. As Figure 10(a) shows, compared with using one warp as usual, our approach achieves higher performance. Moreover, our batched EVD kernel implements the parallelization of the two-sided update, which achieves more than 6 times faster than the sequential one, as shown in Figure 10(b).



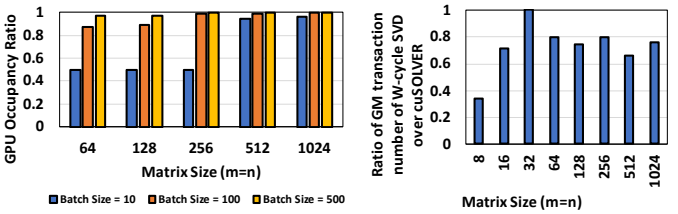
(a) Runtime of Batched SVD (b) Runtime of EVD

Fig. 10: Evaluation on different approaches in W-cycle SVD. Each matrix size is 32×32.

Figure 11(a) shows the GPU occupancy rate of W-cycle SVD. With the batch size enlarging, the GPU occupancy rate of W-cycle SVD increases monotonously, which consists with the fact that the performance speedup becomes larger with the increase of batch size. Further, the occupancy rate of GPU gradually approaches the GPU's peak occupancy rate as batch size increases from 10 to 500.

Locality. W-cycle SVD improves the data locality significantly. Usually, the fewer GM transactions indicate the better data locality of kernels. Figure 11(b) gives the comparison of the overall GM transactions between W-cycle SVD and cuSOLVER. It is clear that W-cycle SVD exchanges fewer data between fast on-chip memory and GM than cuSOLVER, which implies that W-cycle SVD has a better data locality. This is because W-cycle SVD makes better use of SM for its flexible choice of w_h so that most of the calculations are performed in SM.

We should note that the number of GM transactions of cuSOLVER is close to that of W-cycle SVD for the case with $m = n = 32$. We guess that cuSOLVER adopts a static algorithm specially optimized for some matrix size e.g., $m = n = 32$. As Figure 7 shows, W-cycle SVD also achieves lower speedup on case of $m = n = 32$ compared with other cases.



(a) GPU occupancy rate of W-cycle SVD. (b) Ratio of GM transaction number of W-cycle SVD over cuSOLVER (batch size = 100).

Fig. 11: GPU occupancy and GM transaction.

C. Evaluation of Tailoring Strategy

Figure 12 shows the speedup of W-cycle SVD with different tailoring strategies over W-cycle SVD without tailoring. We find that W-cycle SVD with the tailoring strategy achieves $1.2\times$ performance speedup on average compared with W-cycle SVD without tailoring. When the batch size is 10, the tailoring strategy with auto-tuning engine provides around $1.11\times$ speedup. With the batch size enlarging, the performance benefit increases. When the batch size is 500, the performance improvement becomes $1.48\times$ at most.

From Figure 12, we conclude two observations. First, with the increase of either batch size or matrix size, the rate of performance improvement increases. The main reason is that the tailoring strategy exploits more parallelism, ensuring a higher GPU occupancy rate. Second, when the matrix size is large enough, the GPU occupancy rate of W-cycle SVD is close to the GPU's peak occupancy rate (Figure 11(a)). The performance benefit from tailoring strategies is no longer significant.

Furthermore, Table V shows the runtime of W-cycle SVD with different tailoring plans. We compare the tailoring plans provided by the auto-tuning engine with all the plans based on the expertise. In most cases, the auto-tuning approach can find

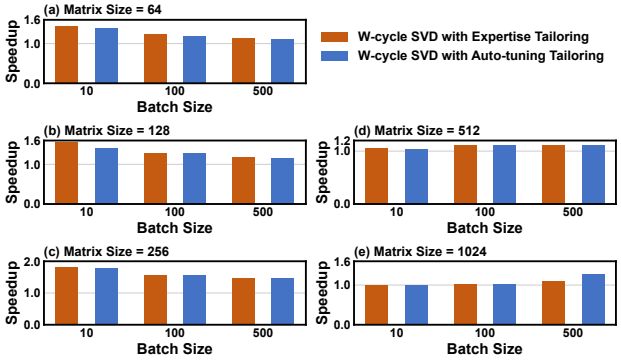


Fig. 12: W-cycle SVD with different tailoring plans for performance improvement over W-cycle SVD without tailoring.

the optimal solution for the tailoring strategy. Even though the auto-tuning solution is sub-optimal, the performance variability is less than 12% compared with the theoretical optimal which is obtained by testing all the candidates. This fact implies that the auto-tuning engine is effective and reliable.

TABLE V: Time (seconds) of W-cycle SVD with different plans.

Tailoring Plan	Matrix Size (m=n)				
	64	128	256	512	1024
$\delta_1 = 32$, $w_1 = 4$	0.0124	0.0639	0.471	3.602	30.58
$\delta_1 = m$, $w_1 = 4$	0.0122	0.0376	0.135	0.453	1.83
$\delta_1 = 32$, $w_1 = 24$	0.0132	0.0304	0.198	1.053	8.21
$\delta_1 = m$, $w_1 = 24$	0.0136	0.0365	0.154	0.421	1.72
$\delta_1 = 32$, $w_1 = 16$	0.0065	0.0272	0.169	1.198	8.55
Auto-tuning	**0.0058**	**0.0272**	**0.152**	**0.421**	**1.59**
Theoretical optimal	**0.0058**	**0.0263**	**0.135**	**0.421**	**1.59**

D. Evaluation of W-cycle SVD with variable matrix sizes

Table VI gives the results of the batched SVD with various matrix sizes. We choose different matrices with variable sizes from SuiteSparse [39], which is one of the most commonly used data sets. The matrices in SuiteSparse are assigned into five groups according to the size metric in the first column of Table VI.

The result shows that W-cycle SVD achieves $2.21 \sim 15.0\times$ speedup over cuSOLVER. It should be noted that, for the second and third groups, W-cycle SVD achieves much higher speedup than the average, because the tailoring strategy significantly improves the parallelism for SVDs in these cases.

TABLE VI: Evaluation of W-cycle SVD with various matrix sizes.

Matrix Size (not larger than)	Batch Size	cuSOLVER (seconds)	W-cycle SVD (seconds)	Speedup
$m, n \leq 32$	46	0.000548	0.000181	3.03
$m, n \leq 64$	85	0.0537	0.00359	**15.0**
$m, n \leq 128$	156	0.287	0.0267	**10.8**
$m, n \leq 256$	243	1.02	0.198	5.18
$m, n \leq 512$	458	6.07	2.75	2.21

E. Sensitivity for GPU Architecture

To demonstrate the portability on GPU architectures, we evaluate W-cycle SVD with the tailoring strategy on different platforms.

Figure 13 shows the applicability on A100 GPU with tensor cores. The performance envelope of W-cycle SVD is pushed further, because the tensor cores significantly accelerate the two batched GEMMs at each level.

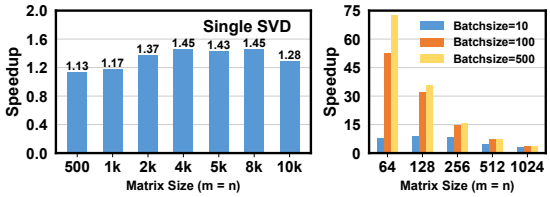
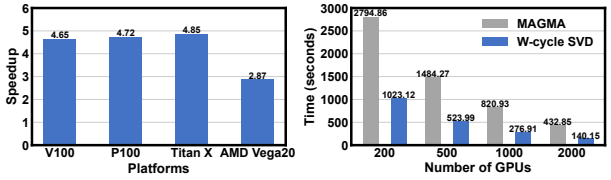


Fig. 13: W-cycle SVD for performance improvement over cuSOLVER on NVIDIA A100 GPU with tensor cores.

For the batched SVD of 100 randomly generated matrices with each size of 512×512, Figure 14(a) shows that, compared with cuSOLVER, W-cycle SVD can achieve $4.56\times$, $4.72\times$ and $4.85\times$ speedups on V100, P100 and GTX Titan X GPUs respectively. Compared with MAGMA, W-cycle SVD achieves $2.85\times$ speedup on AMD Vega20 GPU with HIP runtime v4.5. We have demonstrated that W-cycle SVD achieves a consistent performance speedup on different architectures.



(a) Sensitivity on different GPUs. (b) Data assimilation application.

Fig. 14: W-cycle SVD for performance improvement.

In addition, some ML/AI algorithms also use SVDs with the data type of low-bit float, e.g., bf16. To further extend W-cycle SVD for these cases, the improvement of our design may focus on two aspects. First, the low-bit representation takes less memory and cache space. Hence, larger matrices can be entirely stored in SM, which allows W-cycle SVD to explore the larger w_h and deeper recursion. Second, to exploit the performance potential of tensor cores with low-bit floats, it is necessary to reconfigure the performance model and auto-tuning engine. This will be our future work.

F. Performance Improvement on Real-world Application

Data assimilation is widely applied to the reconstruction of observed historical data for providing initial conditions of numerical atmospheric [40], [41] and oceanic models [36]–[38]. On the latitude-longitude mesh of an oceanic model with 0.1° spatial resolution, the data assimilation on each grid point involves one SVD. SVDs in different points can be batched together, and each matrix size ranges from 50×50 to 1024×1024. Figure 14(b) shows the computation time of data assimilation using W-cycle SVD and MAGMA on a distributed-memory system with AMD Vega20 GPU. W-cycle SVD achieves $2.73 \sim 3.09\times$ speedup compared with MAGMA.

G. Evaluation on the convergence speed and accuracy

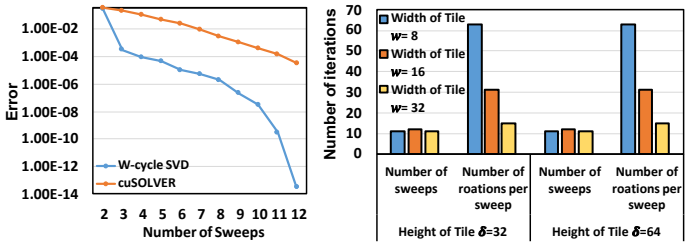
To verify the robustness of W-cycle SVD, we compare the convergence speed and final accuracy of W-cycle SVD and cuSOLVER for a single SVD. In this test, five matrices are chosen from SuiteSparse, as shown in Table VII. For the fixed final accuracy, W-cycle SVD needs fewer sweeps than cuSOLVER. Furthermore, with the increase of the matrix condition number, both the convergence would delay. However, W-cycle SVD still converges faster.

Figure 15(a) confirms that W-cycle SVD also has the advantage on the final accuracy. At any sweep, W-cycle SVD owns a lower error compared with cuSOLVER.

Furthermore, Figure 15(b) shows how the tile size affects the convergence speed. It is clear that the number of rotations per sweep decreases with w_h increasing, which leads to a faster convergence speed. Meanwhile, for a fixed w_h, changing δ_h can not affect the convergence rate.

TABLE VII: Evaluation on the accuracy and convergence speed.

Matrix Feature			Number of sweeps (Error is less than 10^{-12})	
Name	Size	Condition Number	cuSOLVER	W-cycle SVD
ash331	331×104	3.10×10^{0}	8	6
impcol_d	425×425	2.06×10^{3}	15	12
tols340	340×340	2.03×10^{5}	14	10
robot24c1_mat5	404×302	3.33×10^{11}	14	13
flower_7_1	463×393	8.08×10^{15}	28	22



(a) Accuarcy Change. (b) Convergence Change.

Fig. 15: Evaluation on the accuracy and convergence speed. The test matrix is impcol_d in Table VII.

VI. Related Work

SVD is an important operator for many applications in a broad range of domains. Related work focuses on SVD optimization from algorithm level [42]–[44] and architecture level [17], [19], [45].

Batched SVD. Recent decade has witnessed the development of batched SVD on GPUs. To the best of our knowledge, the GPU-CPU hybrid algorithm for batched SVD was first proposed for the detection of quiet targets in underwater acoustic array signal processing [2]. This algorithm focused on the pair generation for the Jacobi method and handled the bi-diagonalization phase of Gram matrices on CPU. Afterward, batched SVD on GPU was studied to generate rank 1 matrices for approximating 2D filters in convolutional neural networks [3], which only focused on how to obtain the largest singular values and the corresponding singular vectors of many small matrices of size less than 15×15. Then, a parallel method for SVD of many matrices on GPUs was proposed for the image mosaic assemble application [4], and one thread within a warp was applied to compute the SVD of a matrix. In recent years, the fine-grained analysis on batched SVD suggested that the matrices of different sizes need different algorithm designs to achieve high performance on GPUs [19]. The related work primarily was size-sensitive [1]–[4], [19], which hardly achieved high data reuse and convergence speed at the same time. In this work, we first propose a W-cycle SVD, which supports two key optimizations: (1) exploiting the data reuse of SM for each SVD, and (2) ensuring the optimal convergence speed for each SVD.

Software Projects. NVIDIA cuSOVLER library [20], [35] is a fast GPU-accelerated package that provides lots of useful linear algebra solvers, such as common matrix factorization routines for dense matrices, a sparse least-squares solver, and an eigenvalue solver, etc. The basic matrix factorization SVD is one of the common routines in cuSOVLER. Meanwhile, cuSOVLER also provides a batched SVD API for programmers. cuSOVLER fully exploits parallel features of Jacobi-based algorithms, and the corresponding SVD solvers represent the state-of-the-art implementation. While it is not open source, cuSOVLER enables a CUDA program to achieve good performance. In addition, Matrix Algebra on GPU and Multi-core Architectures (MAGMA) [22], [35] is a dense linear library for multi-core GPU systems, which is open source. MAGMA group designed linear algebra algorithms and frameworks for hybrid many-core and GPU systems that enable applications to fully exploit the power of hybrid components [46]. MAGMA supports the SVD bi-diagonalization in distributed-memory GPU systems [21]. To the best of our knowledge, our work is the first one to propose a uniform algorithm for batched SVD with various matrix sizes. Compared with cuSOLVER and MAGMA, W-cycle SVD achieves significant performance improvement for bathed SVD on GPUs.

VII. Conclusion

This work presents an in-depth analysis on the performance of batched SVD from the algorithm level. Based on two observations, we develop W-cycle SVD, which is a multilevel algorithm for batched SVD on GPUs. To push the envelope of performance further, we design the efficient batched SVD and EVD kernels, and propose a tailoring strategy to accelerate batched GEMM in SVDs. The experimental results confirm the high performance of W-cycle SVD.

Acknowledgment

The authors would like to thank all anonymous reviewers for their valuable comments and helpful suggestions. The work is supported National Natural Science Foundation of China, under Grant No. 62172391, 61972377, 62032023, T2125013, and International Partnership Program of Chinese Academy of Sciences, under Grant No. 171111KYSB20180011.

REFERENCES

[1] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale," *SIAM Review*, vol. 60, no. 4, pp. 808–865, 2018. [Online]. Available: https://doi.org/10.1137/17M1117732

[2] C. Kotas and J. Barhen, "Singular value decomposition utilizing parallel algorithms on graphical processors," in *OCEANS'11 MTS/IEEE KONA*. IEEE, September 2011, pp. 1–7. [Online]. Available: https://doi.org/10.23919/OCEANS.2011.6107024

[3] H.-P. Kang and C.-R. Lee, "Improving performance of convolutional neural networks by separable filters on gpu," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 638–649. [Online]. Available: https://doi.org/10.1007/978-3-662-48096-0_49

[4] I. Badolato, L. de Paula, and R. Farias, "Many svds on gpu for image mosaic assemble," in *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*. IEEE, October 2015, pp. 37–42. [Online]. Available: https://doi.org/10.1109/SBAC-PADW.2015.22

[5] S. Kudo and Y. Yamamoto, "On using the cholesky qr method in the full-blocked one-sided jacobi algorithm," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer International Publishing, 2018, pp. 612–622. [Online]. Available: https://doi.org/10.1007/978-3-319-78024-5_53

[6] M. Gates, S. Tomov, and J. Dongarra, "Accelerating the svd two stage bidiagonal reduction and divide and conquer using gpus," *Parallel Computing*, vol. 74, pp. 3–18, 2018, parallel Matrix Algorithms and Applications (PMAA'16). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819117301758

[7] I. Yamazaki, S. Tomov, and J. Dongarra, "Stability and performance of various singular value qr implementations on multicore cpu with a gpu," *ACM Trans. Math. Softw.*, vol. 43, no. 2, September 2016. [Online]. Available: https://doi.org/10.1145/2898347

[8] V. Hari, "Accelerating the svd block-jacobi method," *Computing*, vol. 75, no. 1, pp. 27–53, July 2005. [Online]. Available: https://doi.org/10.1007/s00607-004-0113-z

[9] J. Demmel and W. Kahan, "Computing small singular values of bidiagonal matrices with guaranteed high relative accuracy: Lapack working note number 3," *Technical Report*, February 1988. [Online]. Available: https://www.osti.gov/biblio/5039344

[10] ——, "Accurate singular values of bidiagonal matrices," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 5, pp. 873–912, 1990. [Online]. Available: https://doi.org/10.1137/0911052

[11] G. Okša and M. Vajteršic, "Preconditioning in the parallel block-jacobi svd algorithm," *Proceedings of ALGORITMY*, pp. 202–211, 2005.

[12] M. Bečka, G. Okša, and M. Vajteršic, "New dynamic orderings for the parallel one-sided block-jacobi svd algorithm," *Parallel Processing Letters*, vol. 25, no. 02, pp. 1–19, June 2015. [Online]. Available: https://doi.org/10.1142/S0129626415500036

[13] G. Okša and M. Vajteršic, "Parallel one-sided block jacobi svd algorithm: I. analysis and design," Technical report, Salzburg University, Salzburg, Austria, Tech. Rep., June 2007.

[14] R. P. Brent, "Parallel algorithms for digital signal processing," in *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, G. H. Golub and P. Van Dooren, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 93–110. [Online]. Available: https://doi.org/10.1007/978-3-642-75536-1_5

[15] M. R. Hestenes, "Inversion of matrices by biorthogonalization and related results," *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, no. 1, pp. 51–90, 1958. [Online]. Available: https://doi.org/10.1137/0106005

[16] R. P. Brent and F. T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 69–84, 1985. [Online]. Available: https://doi.org/10.1137/0906007

[17] S. Lahabar and P. J. Narayanan, "Singular value decomposition on gpu using cuda," in *2009 IEEE International Symposium on Parallel Distributed Processing*. IEEE, May 2009, pp. 1–10. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5161058

[18] V. Novaković and S. Singer, "A gpu-based hyperbolic svd algorithm," *BIT Numerical Mathematics*, vol. 51, no. 4, pp. 1009–1030, June 2011. [Online]. Available: http://dx.doi.org/10.1007/s10543-011-0333-5

[19] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, "Batched qr and svd algorithms on gpus with applications in hierarchical matrix compression," *Parallel Computing*, vol. 74, pp. 19–33, 2018, parallel Matrix Algorithms and Applications (PMAA'16). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819117301461

[20] C. Toolkit, "Cuda toolkit documentation," 2018. [Online]. Available: http://docs.nvidia.com/cuda/index.html

[21] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "Accelerating the svd bi-diagonalization of a batch of small matrices using gpus," *Journal of Computational Science*, vol. 26, pp. 237–245, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187775031731150X

[22] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *The International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010. [Online]. Available: https://doi.org/10.1177/1094342010385729

[23] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for gpus," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[24] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, January 2012. [Online]. Available: https://doi.org/10.1109/TPDS.2011.311

[25] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning gemm for gpus," in *Computational Science – ICCS 2009*, G. Allen, J. Nabrzyski, E. Seidel, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 884–892. [Online]. Available: https://doi.org/10.1007/978-3-642-01970-8_89

[26] A. Brandt, S. McCormick, and J. Ruge, "Multigrid methods for differential eigenproblems," *SIAM Journal on Scientific and Statistical Computing*, vol. 4, no. 2, pp. 244–260, 1983. [Online]. Available: https://doi.org/10.1137/0904019

[27] H. Xie, "A multigrid method for eigenvalue problem," *Journal of Computational Physics*, vol. 274, pp. 550–561, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999114004379

[28] P. P. M. de Rijk, "A one-sided jacobi algorithm for computing the singular value decomposition on a vector computer," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, no. 2, pp. 359–371, 1989. [Online]. Available: https://doi.org/10.1137/0910023

[29] M. Bečka, G. Okša, and M. Vajteršic, "Dynamic ordering for the parallel one-sided block-jacobi svd algorithm," in *Proceedings of the Conference on Parallel Numerics 2011 (ParNum 11)*. Graz University Press, October 2011, pp. 5–15.

[30] S. Kudo, Y. Yamamoto, M. Bečka, and M. Vajteršic, "Performance analysis and optimization of the parallel one-sided block jacobi svd algorithm with dynamic ordering and variable blocking," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 9, p. e4059, December 2016, e4059 cpe.4059. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4059

[31] B. Zhou and R. Brent, "A parallel ring ordering algorithm for efficient one-sided jacobi svd computations," *Journal of Parallel and Distributed Computing*, vol. 42, no. 1, pp. 1–10, April 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731597913046

[32] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling, "Block reduction of matrices to condensed forms for eigenvalue computations," *Journal of Computational and Applied Mathematics*, vol. 27, no. 1, pp. 215–227, September 1989, special Issue on Parallel Algorithms for Numerical Linear Algebra. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377042789903671

[33] J. Xu, "Iterative methods by space decomposition and subspace correction," *SIAM Review*, vol. 34, no. 4, pp. 581–613, 1992. [Online]. Available: https://doi.org/10.1137/1034116

[34] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, April 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[35] A. Chrzęszczyk and J. Anders, *Matrix computations on the GPU. CUBLAS, CUSOLVER and MAGMA by Example. Version 2017*. Nvidia, December 2017.

[36] A. A. Emerick and A. C. Reynolds, "Ensemble smoother with multiple data assimilation," *Computers and Geosciences*, vol. 55, pp. 3–15, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098300412000994

[37] J. Xiao, S. Wang, W. Wan, X. Hong, and G. Tan, "S-enkf: Co-designing for scalable ensemble kalman filter," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 15—26. [Online]. Available: https://doi.org/10.1145/3293883.3295722

[38] J. Xiao, G. Zhang, Y. Gao, X. Hong, and G. Tan, "Fast data-obtaining algorithm for data assimilation with large data set," *International Journal of Parallel Programming*, vol. 48, no. 4, pp. 750–770, 2020. [Online]. Available: https://doi.org/10.1007/s10766-019-00653-y

[39] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, December 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[40] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "A peta-scalable cpu-gpu algorithm for global atmospheric simulations," in *Proceedings of the 18th ACM SIGPLAN symposium on principles and practice of parallel programming*, ser. PPoPP '13, vol. 48, no. 8. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2517327.2442518

[41] J. Xiao, S. Li, B. Wu, H. Zhang, K. Li, E. Yao, Y. Zhang, and G. Tan, "Communication-avoiding for dynamical core of atmospheric general circulation model," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225140

[42] M. Bečka, G. Okša, and E. Vidličková, "New preconditioning for the one-sided block-jacobi svd algorithm," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer International Publishing, March 2018, pp. 590–599. [Online]. Available: https://doi.org/10.1007/978-3-319-78024-5_51

[43] B. Foster, S. Mahadevan, and R. Wang, "A gpu-based approximate svd algorithm," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds., vol. 7203. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 569–578. [Online]. Available: https://doi.org/10.1007/978-3-642-31464-3_58

[44] S. Li, M. Gu, L. Cheng, X. Chi, and M. Sun, "An accelerated divide-and-conquer algorithm for the bidiagonal svd problem," *SIAM Journal on Matrix Analysis and Applications*, vol. 35, no. 3, pp. 1038–1057, 2014. [Online]. Available: https://doi.org/10.1137/130945995

[45] V. Demchik, M. Bačák, and S. Bordag, "Out-of-core singular value decomposition," pp. 1–17, July 2019.

[46] A. Haidar, T. T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "A framework for batched and gpu-resident factorization algorithms applied to block householder transformations," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, June 2015, pp. 31–47. [Online]. Available: https://doi.org/10.1007/978-3-319-20119-1_3

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

A ARTIFACT DETAILS

A.1 Abstract Summary

W-cycle SVD is a multilevel algorithm for batched SVD on GPUs. W-cycle SVD is size-oblivious, which successfully exploits the data reuse and ensures the optimal convergence speed for multiple SVDs. To push the envelope of performance further, we design the efficient batched SVD and EVD kernels, and propose a tailoring strategy to accelerate batched GEMMs in SVDs. The project we provided includes the full source code of W-cycle SVD program. We also append extra contents for the convenience of those experiments reported in our paper.

A.2 Abstract Checklist

- **Platforms:**
  1. NVIDIA CUDA platforms
  The Platforms we used are:
  - Tesla V100
  - Tesla P100
  - GTX TiTan X
  - Ampere A100

  2. AMD ROCm platforms
  The Platforms we used are:
  - A single AMD Vega20 GPU
  - A cluster with Vega20 GPUs
- **System Details:**
  - 18.04-Ubuntu x86_64 GNU/Linux (V100, P100 and TiTan X)
  - CentOS 7.9 (A100)
  - CentOS 7.6 (AMD Vega20 GPU)

- **Software Dependencies:**
  - GNU Make 4.1
  - CUDA toolkit (tested 10.1, 11.6)
  - nvprof profiling tool
  - gcc/g++ (tested 4.8.5, 7.5)
  - ROCm (tested 3.5, 4.2)
  - Intel oneMKL (tested 2022.1.0)
  - MAGMA (tested 2.5.4)

A.3 Environment Setup

Step 1: Basic environment.
(a) CUDA Platform:
- CUDA toolkit (version more than 10.1) should be installed. The compiler used is nvcc. Extra libraries needed are cuSOLVER, cuBLAS and MAGMA. Here, MAGMA depends on CUDA toolkit and intel oneMKL.

(b) ROCm Platform:
- ROCm toolkit (version more than 4.2) should be installed. The compiler used is hipcc. The extra library needed is MAGMA (hip). Here, MAGMA depends on ROCm toolkit and intel oneMKL.

Step 2: Compile the program.
The project can be accessed on the Github by:
Link: https://github.com/MOLOjl/WCycleSVD
- Use git (http, ssh, etc.) to clone the repository into a local directory. For the four environments on which our artifact is tested, there are 4 branches:
  (i) main_CUDA,
  (ii) test_Tensor_Core,
  (iii) test_HIP,
  (iv) test_Cluster
- Run ***make*** at the root directory, after respectively cloning each branch to the corresponding platform.

Step 3: Prepare necessary data.
For the "main_CUDA" branch, the data are too large to store in the repository on the Github website. Please generate them manually by running the following commands:
- ***unzip data/UF_matrixset.zip***
- ***./test 99***

A.4 Experiments list

This list shows how to reproduce the results of all the experiments in the **revised paper**.

(i) On V100, P100 and GTX TiTan X ("main_CUDA" branch):

(1) Time of one-sided Jacobi methods in different cases. (Fig. 1)
  - Run: ***./test 1***
(2) One-sided Jacobi method for a batched SVD of 100 matrices with each size of 1536×1536. (Fig. 2)
  - Run: ***./test 2***
(3) Different tile sizes for two batched GEMMs at Level 1 of W-cycle SVD with two levels for 100 matrices. (TABLE I)
  - Run: ***./test 3***
(4) W-cycle SVD for improvement over cuSOLVER with matrix size below 32. (Fig. 7)
  - Run: ***./test 4***
(5) Comparison with cuSOLVER using batch size=1 with matrix size between 500 and 10000. (Fig. 8(a))
  - Run: ***./test 5***
(6) W-cycle SVD for performance improvement with matrix size between 64 and 1024. (Fig. 8(b))
  - Run: ***./test 6***
(7) W-cycle SVD for performance improvement over MAGMA. (Fig. 9)
  - Run: ***./test 7***
(8) Time(s) for SVDs of 200 Matrices on P100 GPU. (TABLE IV)
  - Run: ***./test 8***
(9) Comparison between different approaches using one warp or α warps for column rotation tasks. (Fig. 10(a))
  - Run: ***./test 9***

(10) Comparison between the original and parallel two-sided Jacobi methods for bathed EVD. (Fig. 10(b))
- Run: ***./test 10***

(11) GPU occupancy rate of batched SVD. (Fig. 11(a))
- Run: ***./test11.sh***

(12) Ratio of GM transaction number of W-cycle SVD over cuSOLVER. (Fig. 11(b))
- Run: ***./test12.sh***

(13) Improvements of the tailoring strategy. (Fig. 12)
- Run: ***./test 13***

(14) Time(s) of W-cycle SVD with different tailoring plans. (TABLE V)
- Run: ***./test 14***

(15) Evaluation of W-cycle SVD with various matrix sizes, with SuiteSparse matrix set. (TABLE VI)
- Run: ***./test 15***

(16) Sensitivity on different GPUs. (Fig. 14(a))
- Run: ***./test 17***

(17) Evaluation on the accuracy and convergence speed. (TABLE VII)
- Run: ***./test 18***

(18) Evaluation on the accuracy. (Fig. 15(a))
- Run: ***./test 19***

(19) Evaluation on the convergence speed. (Fig. 15(b))
- Run: ***./test 20***

(ii) On A100 ("test_Tensor_Core" branch):

(1) Evaluation on A100 GPU with tensor cores. (Fig. 13)
- Run: ***./test 16***

(iii) On AMD Vega20 GPU ("test_HIP" branch):

(1) Sensitivity on different GPUs. (Fig. 14(a))
- Run: ***./svd***

(iv) On GPU cluster ("test_Cluster" branch):

(1) Data assimilation application. (Fig. 14(b))
- Run: ***sbatch test18.slurm***

The number of GPUs used is defined in the script **test18.slurm**. After the program finished, the result will be written in **test18.o**.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://github.com/MOLOjl/WCycleSVD`
Artifact name: WCycleSVD

Artifact 2

Persistent ID: `10.5281/zenodo.6585881`
Artifact name: WCycleSVD

Reproduction of the artifact with container: We have provided all the source codes and the building guidance on the Github website. Please follow the "README.md" file or the instruction in the artifact details to reproduce our artifact.