Mille-feuille: A Tile-Grained Mixed Precision Single-Kernel Conjugate Gradient Solver on GPUs

Dechuang Yang*, Yuxuan Zhao*, Yiduo Niu*, Weile Jia†‡, En Shao†‡, Weifeng Liu*, Guangming Tan†‡, Zhou Jin*
*Super Scientific Software Laboratory, Dept. of CST, China University of Petroleum-Beijing, China
† State Key Lab of Processors, Institute of Computing Technology, CAS, China
‡ University of Chinese Academy of Sciences, China
Email: *{dechuang.yang, yuxuan.zhao, yiduo.niu}@student.cup.edu.cn, †‡{jiaweile, shaoen}@ict.ac.cn
*weifeng.liu@cup.edu.cn, †‡tgm@ict.ac.cn, *jinzhou@cup.edu.cn

***Abstract*—Conjugate gradient (CG) and biconjugate gradient stabilized (BiCGSTAB) are effective methods used for solving sparse linear systems. We in this paper propose Mille-feuille, a new solver for accelerating CG and BiCGSTAB on GPUs. We first analyze the two methods and list three findings related to the use of mixed precision, the reduction of kernel synchronization costs, and the awareness of partial convergence during the iteration steps. Then, (1) to enable tile-grained mixed precision, we develop a tiled sparse format; (2) to reduce synchronization costs, we leverage atomic operations that make the whole solving procedure work within a single GPU kernel; (3) to support a partial convergence-aware mixed precision strategy, we enable tile-wise on-chip dynamic precision conversion within the single kernel at runtime. The experimental results on an NVIDIA A100 and an AMD MI210 show that the Mille-feuille solver outperforms baseline implementations using the vendor-support cuSPARSE/hipSPARSE as well as two state-of-the-art libraries PETSc and Ginkgo by a factor of on average 3.03x/2.68x, 5.37x, 4.36x (up to 8.77x/7.14x, 16.54x, 15.69x) in CG, on average 2.65x/2.32x, 3.57x, 3.78x (up to 7.51x/6.63x, 16.64x, 11.73x) in BiCGSTAB, on average 3.82x/3.47x (up to 40.38x/47.75x) in preconditioned CG (PCG), on average 1.79x/1.63x (up to 45.63x/44.34x) in preconditioned BiCGSTAB (PBiCGSTAB), respectively.**

***Index Terms*—CG, BiCGSTAB, mixed precision, GPU**

I. Introduction

Iterative solvers [1], particularly a series of Krylov subspace methods [2]–[5], are pivotal in solving large systems of sparse linear equations. Among these methods, conjugate gradient (CG) [6] and biconjugate gradient stabilized (BiCGSTAB) [7] algorithms stand out for their effectiveness in tackling symmetric positive-definite and nonsymmetric or indefinite linear systems, respectively. Consequently, significant efforts have been directed towards accelerating CG and BiCGSTAB methods on various modern parallel computing platforms, such as GPUs [8]–[10] and distributed systems [11]–[13].

In these solvers, a substantial portion of floating-point operations is performed, including sparse matrix-vector multiplication (SpMV), dot product computation, and operation $y = \alpha x + y$ (AXPY). The computational costs of these operations vary with the data types utilized, such as 64-bit double, 32-bit single, 16-bit half and 8-bit minifloat [14], [15]. Different precision levels exhibit distinct trade-offs in terms of space/time complexities and convergence speed [16]. Therefore, the adoption of mixed precision techniques in iterative solvers has received significant attention in recent research efforts [17]–[21].

However, existing mixed precision iterative solvers are typically employed either as a high precision iterative refinement stage after a direct solver [20]–[26], or in preconditioning [27]–[29], or at different levels within algebraic multigrid [30], [31]. Despite their utility, a number of performance-critical factors, such as the distribution of numerical precision, inter-kernel synchronization, and partial convergence of the solution x, are often overlooked in current methods.

We in this work first perform a multiperspective analysis on CG and BiCGSTAB algorithms on GPUs, and then focus on three factors highly related to their execution efficiency: (1) precision-aware space distribution of the values of the nonzeros (Section II-A), (2) high synchronization cost between CUDA kernel calls (Section II-B), and (3) partial convergence of the elements in the solution vector x (Section II-C). On top of the analysis, we also provide three findings to motivate the algorithm design of our Mille-feuille solver.

According to the three findings, we propose Mille-feuille, an effective high performance CG and BiCGSTAB solver that (1) stores a sparse matrix in tiles with different initial precisions, (2) uses just one kernel call to complete SpMV, dot product, and AXPY in an entire procedure, instead of calling a number of individual CUDA kernels, and (3) changes the precision of a column of tiles in on-chip memory when the corresponding elements in the solution vector x are partially convergenced.

In our experiments, we use two GPUs, an NVIDIA A100 and an AMD MI210, and benchmark all suitable matrices, i.e., 230 symmetric positive-definite matrices for CG and 686 nonsymmetric or indefinite matrices for BiCGSTAB, from the entire SuiteSparse Matrix Collection [32]. We compare our algorithm with the baseline implementations of CG and BiCGSTAB using cuSPARSE v12.0/hipSPARSE v2.3.8, as well as with two state-of-the-art libraries PETSc v3.20 [33] and Ginkgo v1.7.0 [34]. The experimental results demonstrate that our Mille-feuille outperforms cuSPARSE/hipSPARSE, PETSc, and Ginkgo by a factor of 3.03x/2.68x, 5.37x, 4.36x in CG; 2.65x/2.32x, 3.57x, 3.78x in BiCGSTAB; 3.82x/3.47x in preconditioned CG (PCG); and 1.79x/1.63x in preconditioned

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

BiCGSTAB (PBiCGSTAB), respectively.

In this work, we make the following contributions:

- We analyze the CG and BiCGSTAB methods and give three findings related to the use of mixed precision for high performance.
- We develop a tiled format that prepares tile-grained mixed precision and dynamic precision switch according to partial convergence at runtime.
- We design a single-kernel method to avoid synchronization costs between GPU kernel calls and to better use on-chip memories for inner-kernel precision conversion.
- We propose a new solver called Mille-feuille that exploits the above optimizations, and significantly outperforms existing work on NVIDIA and AMD GPUs.

II. Background, Analysis and Findings

A. Distribution of Numerical Precision

To take advantage of mixed precision, it is important to first set the appropriate floating point data type to store the input matrix A. There can be a variety of storage strategies depending on the target granularity. A straightforward way is to store multiple complete instances of A with different precisions, and use them in the best stages in a linear solver [19], [30], [35]. It is also possible to divide A into multiple matrices (ones with higher precision and the others with lower precision), and to call a kernel multiple times for different precisions to complete a procedure [18], [36].

However, it is also possible to move forward for a finer-grained observation. Specifically, we investigate the proper initial precision on the nonzero-level and visualize the 'enough good' precision of each nonzero element of three example matrices in Figure 1. As can be seen, four colors are used to present the 'enough good' precision for each nonzero (blue, green, purple and red for FP64, FP32, FP16 and FP8, respectively), and the matrices thus demonstrate diverse distributions of numerical precision.

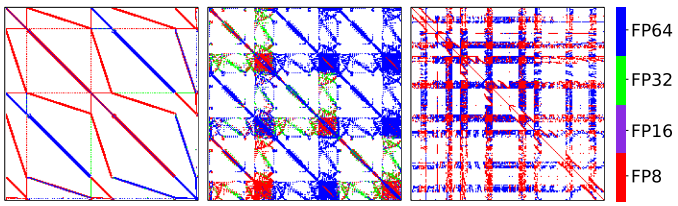


Fig. 1: Three sparse matrices containing nonzeros stored in their 'enough good' precisions.

Our judging criterion of whether a nonzero can be adequately represented in a low precision is shown as follows. We first store each nonzero in four data types, and compute the loss between three lower precisions (i.e., FP32, FP16 and FP8) and the FP64. If the losses of FP32, FP16 and FP8 are less than 10^{-15} (i.e., the decimal digits of precision of FP64), it indicates that the precision FP32, FP16 or FP8 is 'good enough' to store the nonzero. In particular, if three losses of FP32, FP16 and FP8 are all below 10^{-15}, the nonzero will be stored in the lowest possible precision.

Back to Figure 1, calculated by using the criterion mentioned above, the three matrices have very different distributions of numerical precision. That is, most nonzeros of the matrix 'garon2' can be presented with a precision FP16 or FP8, instead of FP64. The matrix 'nmos3' has obvious large block patterns, but half of the blocks should use FP64, and the other half could basically be in FP8. As for the circuit matrix 'ASIC_320k', the small blocks are often in FP8, and the row/column rectangular connections between them should be mostly in FP64.

We thus have the **Finding 1**: There exists a potential to leverage the finer-grained precision distribution to better save the nonzeros of a matrix.

B. Synchronization Costs in CG and BiCGSTAB

We further analyze the algorithm procedure of the CG and BiCGSTAB methods. As shown in Algorithms 1 and 2, SpMV, dot product, and AXPY functions are called in a certain order.

Algorithm 1 A pseudocode of CG.

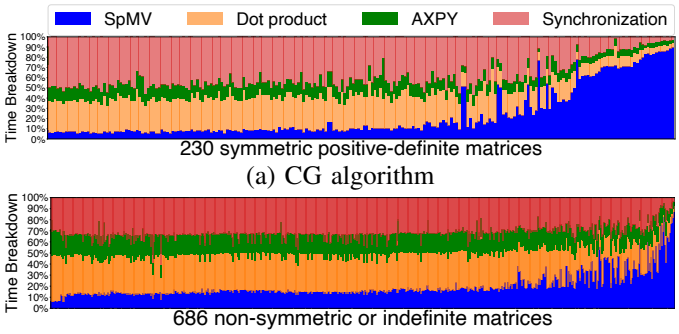
1: Initialize x_0
2: $r_0 = b - Ax_0$
3: $p_0 = r_0, \varepsilon = 10^{-10}$
4: **for** j = 0 **to** *maxiter* **until** $\|r_j\|_2 < \varepsilon$ **do**
5: $\mu = Ap_j$ //SpMV
6: $a_j = (r_j, r_j)/(\mu, p_j)$ //Dot product
7: $x_{j+1} = x_j + a_j p_j$ //AXPY
8: $r_{j+1} = r_j - a_j \mu$ //AXPY
9: $\beta_j = (r_{j+1}, r_{j+1})/(r_j, r_j)$ //Dot product
10: $p_{j+1} = r_{j+1} + \beta_j p_j$ //AXPY
11: **end for**

Algorithm 2 A pseudocode of BiCGSTAB.

1: Initialize x_0
2: $r_0 = b - Ax_0$, r_0^*=r_0
3: $p_0 = r_0, \varepsilon = 10^{-10}$
4: **for** j = 0 **to** *maxiter* **until** $\|r_j\|_2 < \varepsilon$ **do**
5: $\mu = Ap_j$ //SpMV
6: $a_j = (r_j, r_0^*)/(\mu, r_0^*)$ //Dot product
7: $s_j = r_j + \alpha_j \mu$ //AXPY
8: $\theta = As_j$ //SpMV
9: $\omega_j = (\theta, s_j)/(\theta, \theta)$ //Dot product
10: $x_{j+1} = x_j + \alpha_j p_j + \omega_j s_j$ //AXPY
11: $r_{j+1} = s_j - \omega_j \theta$ //AXPY
12: $\beta_j = \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} * \frac{\alpha_j}{\omega_j}$ //Dot product
13: $p_{j+1} = r_{j+1} + \beta_j (p_j - \omega_j \mu)$ //AXPY
14: **end for**

Typically, there exist synchronization overheads between the three kernels. We test all suitable matrices, i.e., 230 for CG and 686 for BiCGSTAB, from the SuiteSparse Matrix Collection [32], and list the costs of the three kernels and the synchronization between them in Figure 2. As can be seen, synchronization often accounts for over 30% of the runtime.

We then give the **Finding 2**: Reducing the synchronization overhead between kernels will likely further optimize the solver performance.



(a) CG algorithm

(b) BiCGSTAB algorithm

Fig. 2: Runtime breakdown of CG and BiCGSTAB methods.

C. Partial Convergence of the Solution Vector

Here, we discuss the partial convergence of the solution vector x. Taking the CG method in Algorithm 1 as an example, when $\|r_j\|_2$ is less than a convergence threshold ε (line 4), the iteration stops. The r_j is calculated based on μ (line 8), which is the output of SpMV $\mu = Ap_j$ (line 5 and Figure 3). In this operation, the matrix A is unchanged, and only the input vector p_j affects the result. Typically, when the values of p_j are numrically small enough, the change of the values of μ will also be small, leading to gradual convergence.

Figure 3 plots the SpMV operation related to possible partial convergence. For particularly small elements (normally smaller than the convergence threshold) in p_j, the nonzeros in the corresponding columns in A no longer need high precision, and could even bypass the multiplications with the elements in p_j.

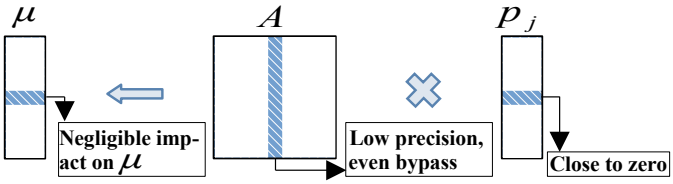


Fig. 3: The small enough elements in p_j need only low precision nonzeros in A or even bypass SpMV on the columns.

As the values of p_j largely determine the convergence of the CG method, we further visualize the changing process of p_j of three representative matrices in Figure 4. As can be seen, from left to right, the values in p_j become progressively smaller (in five ranges, purple, blue, green, orange and red, representing ∞ to 0), until the iterations converge. The three matrices have diverse convergence processes: the matrix 'bcsstm37' works pretty normal, 'Muu' demonstrates an early convergence (i.e., many elements become orange long before the end), and 'm3plates' has a large portion of elements remaining unchanged (orange) from the very beginning. The example means that the recognition of partial convergence has an opportunity to save some calculations in the SpMV operation.

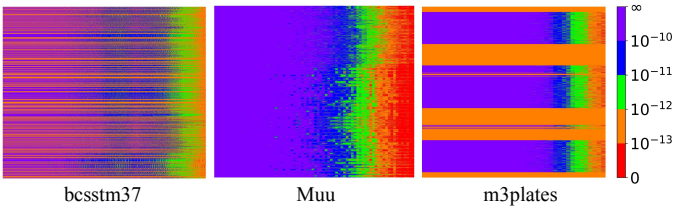


Fig. 4: An example of the diverse processes of convergence of running CG on three matrices. The x-axis shows iteration steps, while the y-axis presents the complete elements in p_j.

We finally obtain the **Finding 3**: For the very small elements in p_j, the nonzeros on the corresponding columns in A may just need lower precision, even bypass the calculations.

III. Mille-feuille

A. Overview

According to the three findings, we propose the Mille-feuille solver for high performance CG and BiCGSTAB with tiled-grained mixed precision (corresponding to the Finding 1) and a single-kernel implementation (corresponding to the Finding 2) considering the partial convergence (corresponding to the Finding 3) on GPUs.

We first partition the entire input matrix into a number of sparse tiles of the same size and leverage a two-level sparse format to store both inter-tile and intra-tile information. For each tile, we store the nonzeros with different precision depending on their initial values. Section III-B will introduce the sparse tile data structure in detail.

Then, we merge the separate CUDA kernels in the naive version into a single kernel, and all nonzeros are loaded into on-chip memory only once and reused in the iterations. Thus, the off-chip load will be minimized, and the synchronization costs will be removed. Section III-C will detail the integration of various computations into a single kernel.

In addition, we consider the use of the elements in the solution vector x partially converged (that is, p_j gets small enough) during the iteration. In this situation, when performing SpMV, the corresponding nonzeros in A could work in a lower precision, or even be bypassed. This is done in tile-grained, by using our data structure. Section III-D presents our strategy.

B. Tile-Grained Storage Structure

Here, we introduce a fine-grained 2D mixed precision storage strategy, which extends the tiled storage formats proposed in [37]–[40]. Specifically, we first partition a sparse matrix into tiles of the same size (16-by-16 is used in this work) and store them using a two-level sparse format. An example is shown in Figure 5.

The high-level storage captures the inter-tile information in a COO style, which includes five arrays: `TileRowidx`, `TileColidx` and `TilePrec` of size $tilenumA$ (where $tilenumA$ is the number of tiles) store the row and column indices, as well as the precision (FP64, FP32, FP16 or FP8,

see Section II-A for the selection criterion) of the tiles, respectively; also, `TileNnz` and `Nonrow` of size $tilenumA + 1$ store the offsets of the number of nonzeros and non-empty rows in the tiles, respectively. The COO style is used to ensure load balancing, as each CUDA warp will compute a tile in our single-kernel implementation.

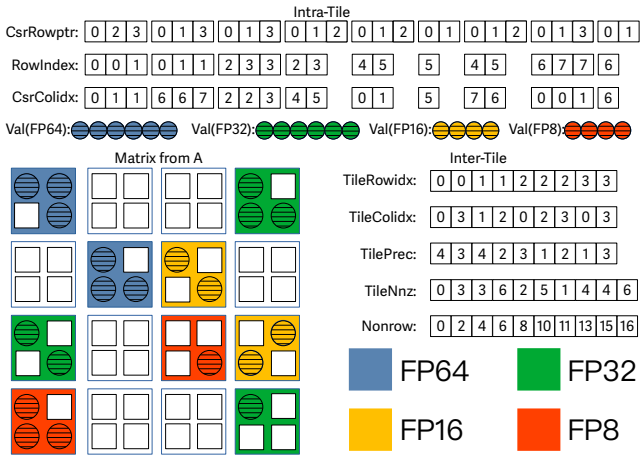


Fig. 5: An example sparse matrix A of size 8-by-8 is stored in nine sparse tiles of size 2-by-2. The high-level structure (inter-tile) consists of five arrays and the precision can be FP64, FP32, FP16 or FP8, and the low-level structure (intra-tile) comprises four arrays.

For the low-level storage, we use the CSR style to record the intra-tile information, which includes three standard arrays `CsrRowptr`, `CsrColidx` and `Val`, as well as an additional array `RowIndex` of size $rownumA$ (the sum of the number of non-empty rows within the tiles) saving the row index of each non-empty row in the tile. The `RowIndex` array, working together with the `Nonrow` array, is designed to avoid traversing empty rows within a tile during the SpMV operation.

C. CG and BiCGSTAB within a Single Kernel

To remove the synchronization costs between different CUDA kernels, we propose a single kernel scheme using atomic operations to resolve dependencies between data and operations. In this way, we consolidate all computations within a single kernel, thereby avoiding costs for kernel barrier synchronization. For brevity, we take CG as the scenario to explain the single-kernel strategy.

We first need to distribute the computational tasks (i.e., operations on sparse tiles) to the CUDA warps. For operations on matrix (SpMV) and vector (dot product and AXPY), we specify two different strategies to assign to warps, respectively. For SpMV, mainly to ensure load balancing, we consider both the number of nonzeros and the number of tiles, and assign the workload to each warp. Specifically, we iterate through the tiles of the matrix, keeping track of the number of nonzeros and tiles allocated to the current warp. If they do not exceed the maximum number of nonzeros and tiles per warp, we assign the sparse tile to the current CUDA warp; otherwise, we assign it to a new warp. For the dot product and AXPY, we dynamically determine the workload for each warp based on the number of vector segments. When the number of segments in a vector does not exceed the maximum number of warps, we assign one warp to deal with each segment. Otherwise, we distribute multiple vector segments to a warp based on the length of the vector and the number of warps available.

Once the workload for each warp has been allocated, we load the matrix into shared memory before the kernel starts, and then reuse it in the subsequent iterations. When the number of nonzeros in the matrix is less than the resources available in shared memory, all nonzeros within each warp will be loaded into shared memory. Otherwise, we will load the nonzeros that do not exceed the maximum shared memory resources within warp into shared memory and put the rest into global memory. In this scenario, we utilize an array to track the number of tiles loaded into shared memory within each warp, enabling distinguish the memory locations of tiles in each warp. When the matrix contains a large number of nonzeros, most of which must be stored in global memory, and the overhead of the global memory accesses outweighs the performance benefits of a single kernel, we revert back to a multi-kernel execution approach (See Figures 8 and 9, on the left of the threshold at 10^6 nonzeros on the x-axis, we run the single-kernel scheme, on the right, the multi-kernel classic method is called).

To realize barrier synchronization without calling multiple kernels, the critical step involves constructing several arrays in global memory to define dependencies that can be resolved within a single kernel, and enabling atomic operations to schedule warps to execute tasks of different operations. We categorize the CG algorithm into four parts based on the dependencies between data and operations, as shown in Figure 6 (Steps A, B, C and D). The dependencies that arise between each step are defined by three arrays in the global memory.

Specifically, we construct an array d_s to resolve the dependencies between SpMV (in Step A) and dot product operation (in Step B). The array d_s retains the remaining workload count of tiles that require the execution of SpMV within the same row tile, respectively, to indicate the beginning of the following dot product. Step B uses the result vector of SpMV to perform the dot-product operation. Therefore, only if all non-empty tiles within the same row tile of the matrix A have completed the SpMV operation, warps can start to work in Step B. The array d_d is used to solve the dependencies between the dot product (in Steps B and C) and the scaling operation (in Steps C and D). Since the scaling operation can only be performed after the dot product is completed, this array tracks the number of warps that have completed the dot product operation for the vector segments. We also use array d_a to resolve the dependencies between the end of the current iteration (in Step D) and the beginning of the next iteration (in Step A). It tracks how many warps have completed the AXPY operation for the vector segment. Only when the AXPY operation on all warps is completed at Step D, the algorithm

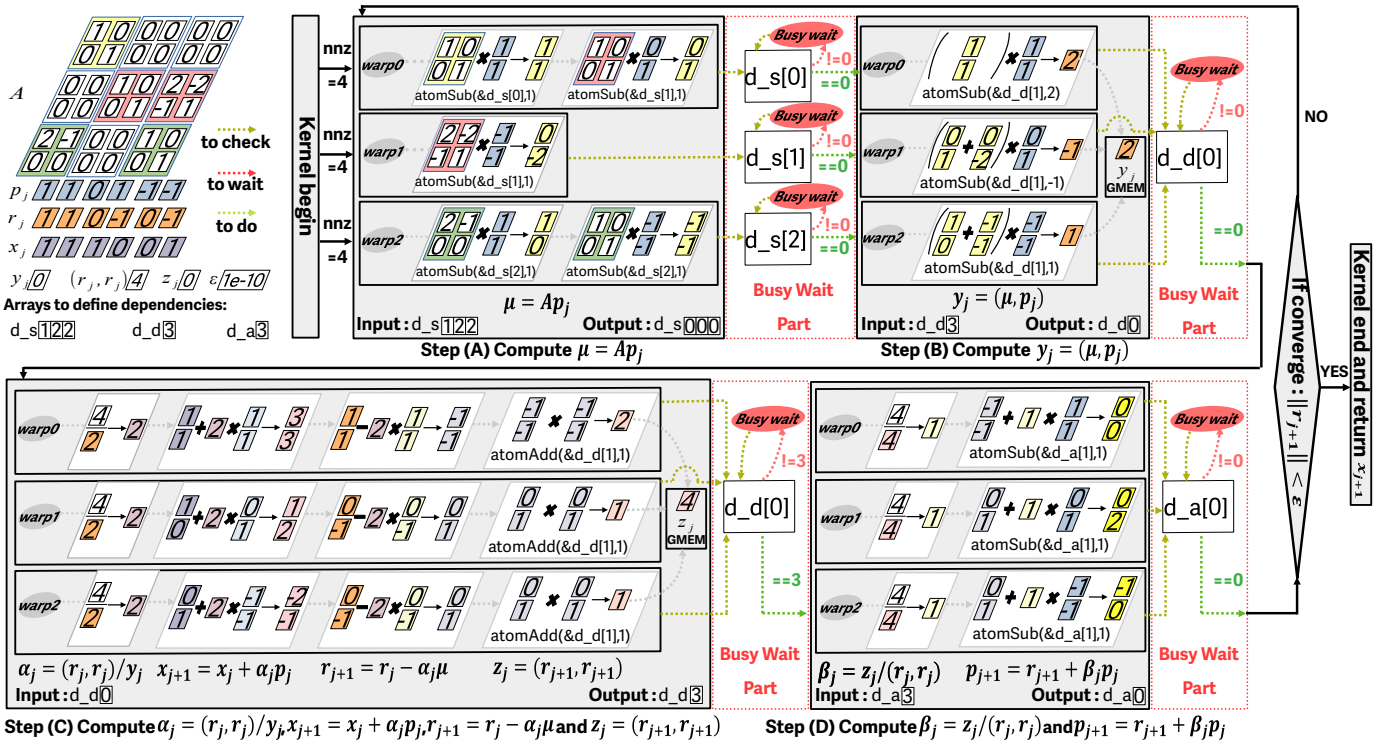


Fig. 6: An example of our proposed single kernel CG method. The tiles for matrix A are first loaded into share memory and distributed to each warp with a similar number of nonzeros and tiles. The computation is divided into four steps (i.e., A, B, C and D). We construct three arrays $d\_s[]$, $d\_d[]$ and $d\_a[]$ in global memory to resolve the data dependencies between each step. Initialize the dependency arrays $d\_s[] = [1, 2, 2]$ (represents how many tiles need to perform SpMV for each row block at Step A), $d\_d[] = [3]$ (represents how many dot product tasks need to perform in steps B and C, respectively), and $d\_a[] = [3]$ (represents how many AXPY operations need to perform in Step D). Each time a task is completed, its corresponding position in dependency array will be decreased atomically by 1. After the warp launches, each warp performs all its tasks in the current step. If all tasks for current Step have been finished at each warp, the warp will check the dependency arrays in global memory to determine whether to wait in the current step or proceed to the next step, as depicted in the Busy Wait Part.

can move to the next iteration.

Next, we show the scheduing details. As plotted in Figure 6, for a 6x6 matrix with five tiles, the matrix has three row tiles. So the d_s array has a length of 3 with initial values 1, 2, 2 according to the number of tiles in each row tile.

Step A. The non-empty tiles of matrix A are firstly distributed to each warp in a load balanced manner to perform tiled-level SpMV (tiles in the same row tile are distributed to different warps). When the SpMV operation of a tile is completed, the value of the corresponding position of d_s is atomically subtracted by 1. After the SpMV tasks in a warp have all been done, this warp checks if its corresponding value in the d_s array is 0. If so, it means that the result vector of SpMV in this row tile has been obtained and this warp can use it to perform the dot product operation for Step B; otherwise, it must busy wait for other tiles in the same row tile to complete their SpMV operations (lines 3-11 in Algorithm 3).

Step B. Each warp performs dot product operations of μ and p_j for the allocated segmented vector. To further improve efficiency, we accumulate their resultant values in shared memory and then accumulate the results into y_j through block-level reduction operations. When a warp completes its dot product tasks, the value in d_d is atomically subtracted by 1 (initial value is 3 in this example). Then each warp checks whether the value of d_d is 0. If so, it indicates that all dot product operations are complete and the final y_j has been obtained, then algorithm moves to Step C, where y_j is divided by (r_j, r_j) to obtain the argument α_j. Otherwise, it will continue to wait for other vector segments to perform the dot product operation (lines 12-18 in Algorithm 3).

Step C. Similar to Step B, when a warp completes the dot product tasks, the value in d_d is atomically added to 1 (initialized from 0) and the results are then reduced to z_j. Each warp checks whether the value of d_d is equal to the number of warps. If so, the dot product operation is completed and z_j can be divided by (r_j, r_j) in Step D to get the parameter β_j; otherwise, it will continue to wait for other warps to perform the dot product (lines 19-26 in Algorithm 3).

Step D. Each warp executes the AXPY operation on allocated vector segments and decreases the value of d_a (initial value is 3) by 1 atomically when the tasks are completed. Each warp verifies if the value of d_a is 0. If so, the ongoing

Algorithm 3 A pseudocode of CG within a single kernel

```
1: for j from 0 to maxiter until convergence do
2:     lane_id = (warp_size - 1) & threadIdx.x
3:     for i from 0 to tilenum in parallel do
4:         row_tile = TileRowidx[i]
5:         for ri from Nonrow[i] to Nonrow[i+1] in parallel do
6:             accumulate inner-product of i^th non-empty row to sum
7:             atomicAdd(u[row_tile × tilesize + RowIndex[ri]], sum)
8:         end for
9:         if lane_id == 0 then atomicSub(d_s[TileRowidx[i]], 1) end if
10:    end for
11:    while d_s[warp_id] ≠ 0 do threadfence() end while
12:    for k from 0 to tilesize in parallel do
13:        index = warp_id × tilesize + k
14:        s_y[warp_id] += u[index] * p[index]
15:    end for
16:    perform block-level reduction for s_y and store the result in y_j
17:    if lane_id == 0 then atomicSub(d_d[0], 1) end if
18:    while d_d[0] ≠ 0 do threadfence() end while
19:    update a_j x and r
20:    for k from 0 to tilesize in parallel do
21:        index = warp_id × tilesize + k
22:        s_z[warp_id] += r[index] * r[index]
23:    end for
24:    perform block-level reduction for s_z and store the result in z_j
25:    if lane_id == 0 then atomicAdd(d_d[0], 1) end if
26:    while d_d[0] ≠ warp_num do threadfence() end while
27:    update p
28:    if lane_id == 0 then atomicAdd(d_a[0], 1) end if
29:    while d_a[0] ≠ warp_num do threadfence() end while
30: end for
```

iteration concludes; otherwise, it awaits the calculation of AXPY for other vector segments. Afterwards, all warps must verify whether the residuals satisfy the convergence criterion. If not, initialize the values of the three dependency arrays and proceed to the next iteration; otherwise, terminate the execution (lines 27-29 in Algorithm 3).

Furthermore, we extend our approach to CG and BiCGSTAB solvers preconditioned with incomplete LU factorization. With the addition of preconditioning steps, such as solving $Mz = r$, it is necessary to incorporate sparse triangular solve (SpTRSV) kernel during the iterative process.

We apply the recursive block SpTRSV algorithm [41] to our multi-kernel method. To be specific, we recursively divide a triangular matrix into two smaller triangular blocks and one square block. The SpTRSV operation is utilized for the triangular blocks, and SpMV is applied to the square blocks. This approach enhances data locality and boosts parallelism through the SpMV operations.

D. Partial Convergence-Aware Mixed Precision Strategy

We further introduce a partial convergence-aware strategy that dynamically switches precisions of the nonzeros of A in a tile-gained way during iterations. Specifically, by assessing the convergence indicated by the elements (16 elements as a basic working unit, aligned to the 16-by-16 tiles) in the vector p_j (i.e., the input vector in SpMV, see Section II-C) at each iteration step, we can convert the current precision of the tiles on the corresponding column to a lower one within the shared memory, or bypass those tiles if needed.

According to the previous analysis (Section II-C), it is evident that multiple elements of x may have converged during the iteration process. When partial elements of the solution vector tend to converge (i.e., residual is less than a tolerance), their corresponding values in the vector p_j typically tend to be less than the convergence threshold. Therefore, in SpMV, the multiplication of these elements with the corresponding tiles in matrix A does not have a significant impact on the subsequent iteration process. We utilize this property to design a faster tile-grained mixed precision SpMV.

We assume that the convergence threshold is ε. If elements in p_j are less than $\varepsilon * 10^{-3}$, we bypass their computation directly. If elements fall within the intervals [ε*10^{-3}, ε*10^{-2}), [ε*10^{-2}, ε*10^{-1}) or [ε*10^{-1}, ε), we reduce the computational precision of the corresponding tiles to FP8, FP16 or FP32, respectively. An example is plotted in Figure 7.

Algorithm 4 A pseudocode of convergent elements retrieval

```
1: thresholds = [ε * 10^-3, ε * 10^-2, ε * 10^-1, ε]
2: flag = [0,0,0,0]
3: vis_flag[warp_id]=0
4: for k =0 to tilesize in parallel do
5:     index=warp_id × tilesize + k
6:     for u = 0 to 3 do
7:         if p_j[index] < thresholds[u] then
8:             atomicAdd(flag[u], 1)
9:         end if
10:    end for
11: end for
12: for u = 0 to 3 do
13:    if flag[u] == tilesize then
14:        vis_flag[warp_id]=u + 1
15:        break
16:    end if
17: end for
```

1) Convergent elements retrieval scheme: To correspond with the tiled structure of A, we divide the vector p_j into several segments, with $tilesize$ denoting the segment length. Specifically, we utilize the array $flag$ of length 4 to keep track of the number of convergent elements located in four different threshold intervals within each segment, respectively. If the element in p_j is less than a certain threshold, we add one to the value of the corresponding position in the array $flag$. Next, we traverse the array $flag$ in turn (lines 4-11 in Algorithm 4). If the value in $flag$ is equal to $tilesize$, it indicates that this segment can be computed with a specific lower precision or bypass (lines 12-17 in Algorithm 4).

2) Dynamic precision adjustment with on-chip conversion: After convergent elements in p_j are retrieved (the precision type of each segment is stored in the array vis_flag), we execute a dynamic precision SpMV. As shown in Algorithm 5 and Figure 7, for each iteration, we first obtain the column index and initial precision type for each tile. Next, we obtain the tile's corresponding value in the vis_flag array (we use 0 - 4 to represent FP64, bypass, FP32, FP16, FP8, respectively) based on the column index and determine whether the tile is bypassed or not (lines 2-3 in Algorithm 5). If the tile needs to be computed, we further determine the computation precision based on both the precision of the segment in the vector p_j (vis_flag) and the initial tile precision type ($TilePrec$) (lines 4-14 in Algorithm 5).

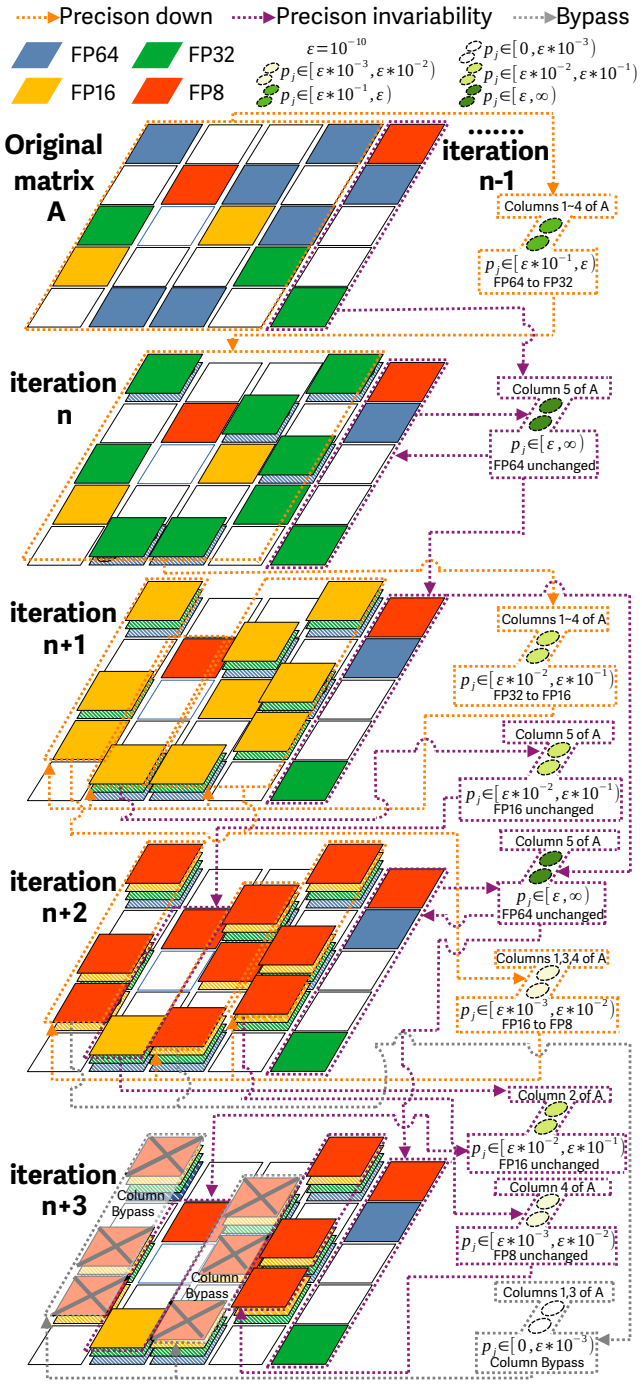


Fig. 7: An example of SpMV in our Mille-feuille multiplying a 10-by-10 sparse matrix, stored as sparse tiles of size 2-by-2, with a vector of length 10. The four different colored tiles indicate the precision of FP64, FP32, FP16 and FP8, respectively, and a cross on red block means bypassed. The three different colored arrows indicate decreasing precision, unchanged precision, and bypass, respectively. The precision of a tile is determined on the basis of five different ranges of values of elements in the vector p_j at each iteration. (Note that we name our work Mille-feuille mainly because the multi-sheet form of the tile-grained mixed precision looks like Mille-feuille the cake.)

Algorithm 5 A pseudocode of mixed precision SpMV

```
1: for i = 0 to tilenumA in parallel do
2:     v_f = vis_flag[TileColidx[i]], b_p= TilePrec[i]
3:     if v_f != 1 then
4:         Use different precision based on the value of v_f and b_p.
5:         nnz_offset = TileNnz[i] , row_tile = TileRowidx[i]
6:         u_offset=row_tile × tilesize
7:         for ri =Nonrow[i] to Nonrow[i+1] in parallel do
8:             sum = 0
9:             for rj = CsrRowPtr[ri] to CsrRowPtr[ri + 1] do
10:                index = nnz_offset+ rj
11:                sum +=p_j[CsrColidx[index]] × s_data[index]
12:            end for
13:            atomicAdd(u[u_offset + RowIndex[ri]], sum)
14:        end for
15:    end if
16: end for
```

If the precision in vis_flag is lower than the initial precision $TilePrec$, we convert the tile precision in shared memory to a lower one (e.g., the second tile in the second column decreases from its initial precision of FP64 to FP32 and then FP16 for the first two iterations in Figure 7). Otherwise, we maintain its initial precision unchanged (e.g., the first tile in the second column in Figure 7). Note that our precision conversion occurs only once in on-chip memory; thereafter, the low-precision values stored in shared memory can be reused. This approach helps us avoid the costs associated with accessing global memory or running precision conversion at each iteration.

Figure 7 illustrates the dynamic evolution of the precision of individual tiles within A, as well as p_j, across four iterations. The original matrix A has seven tiles with precision FP64, three tiles with precision FP32, two tiles with precision FP16 and one tile with precision FP8. For the n_{th} iteration, the values of p_j for columns 1 to 4 are within the range $[\varepsilon \times 10^{-1}, \infty)$. Therefore, six tiles in FP64 are converted to FP32. For the $n+1_{th}$ iteration, the vector values p_j for columns 1 to 4 decrease to the interval $[\varepsilon * 10^{-2}, \varepsilon * 10^{-1})$ and therefore eight tiles in FP32 are converted to FP16. For the $n+2_{th}$ iteration, the values in p_j for columns 1, 3 and 4 decrease to the interval $[\varepsilon * 10^{-3}, \varepsilon * 10^{-2})$ and therefore nine tiles in FP16 are converted to FP8. For the $n+3_{th}$ iteration, the values in p_j for columns 1 and 3 decrease to the interval $[0, \varepsilon * 10^{-3})$ and therefore six tiles are bypassed.

IV. Experimental Results

A. Experimental Setup

Our experimental platform includes two GPUs: an NVIDIA A100 (driver version 525.85.12, CUDA version 12.0) and an AMD MI210 (driver version 6.2.4, RoCM version 5.7.3). We first compare our algorithm with the baseline implementations of CG and BiCGSTAB using the routine `cusparseSpMV()`, `cusparseSpSV_solve()` in cuSPARSE v12.0 and `hipsparseSpMV()`, `hipsparseSpSV_solve()` in hipSPARSE v2.3.8 (both using the CSR format) and `cublasDdot()` in cuBLAS v12.0 and `hipblasDdot()` in hipBLAS v2.3.8 on the A100 and MI210 cards, respectively. Also, we compare our algorithm with two

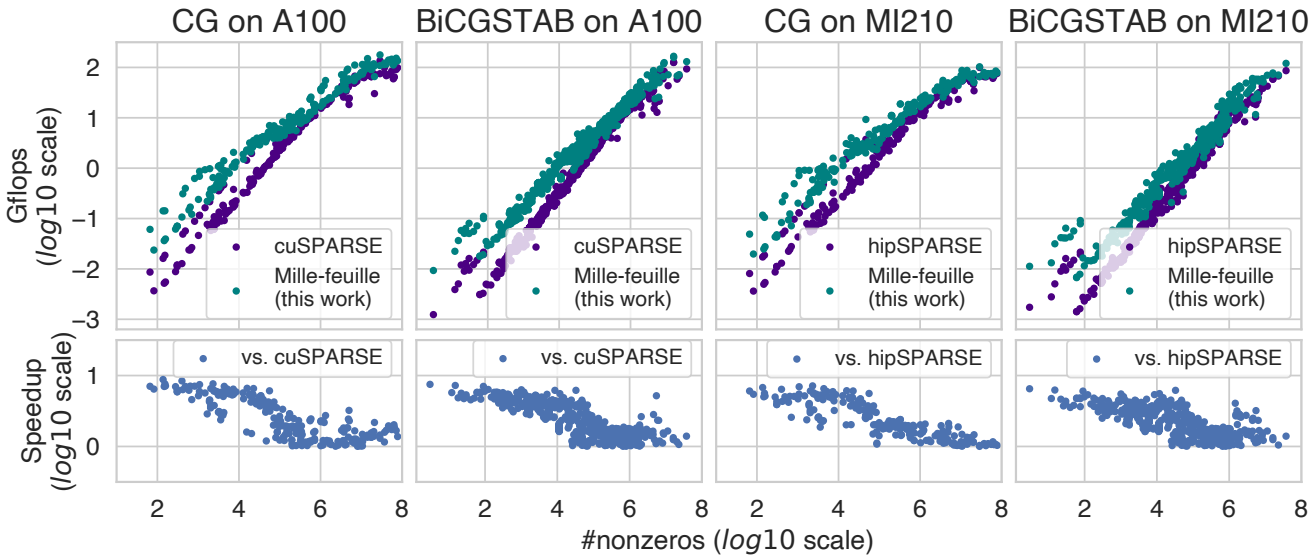


Fig. 8: Performance comparisons between Mille-feuille and the baseline implemented by calling cuSPARSE and hipSPARSE of CG and BiCGSTAB on the NVIDIA A100 and AMD MI210 GPUs with 100 iterations, respectively.

state-of-the-art libraries running on the NVIDIA GPU, with the routine `KSPSolve()` in PETSc v3.20 [33] and `gko::solver::Cg` and `gko::solver::Bicgstab` in Ginkgo v1.7.0 [34] on A100. Our Mille-feuille solver supports both CUDA and HIP and is evaluated on GPUs of the two vendors. The specifications of the two GPUs and the five algorithms tested are listed in Table I.

NVIDIA and AMD GPUs	Five methods tested
(1) A100 (Ampere) PCIe,	(1) cuSPARSE and cuBLAS v12.0 on NVIDIA
6912 CUDA cores @ 1410 MHz,	(2) hipSPARSE and hipBLAS v2.3.8 on AMD
40 GB, B/W 1555 GB/s	(3) PETSc v3.20 on NVIDIA
(2) MI210 (CDNA2) PCIe,	(4) Ginkgo v1.7.0 on NVIDIA
6656 stream processors @ 1700 MHz,	(5) Mille-feuille (this work)
64 GB, B/W 1638 GB/s	on both NVIDIA and AMD

TABLE I: Information of the test platforms and algorithms.

In our experiment, the stopping criterion for convergence is that the relative residual must be less than 10^{-10} and the maximum number of iterations allowed is 1,000. The right-hand side vector is set to be the product of the input matrix and an all 1.0 vector, and the initial value of the solution vector is set to zero (i.e., no preconditioner is used). The precision of other works is set to FP64.

Our experimental dataset includes all 230 symmetric positive-definite matrices for CG and 686 nonsymmetric or indefinite matrices for BiCGSTAB, from the entire SuiteSparse Matrix Collection [32].

B. Comparison over Baseline, PETSc and Ginkgo

We first compare the time taken for 100 iterations between Mille-feuille and the baseline on CG and BiCGSTAB methods calling SpMV in cuSPARSE/hipSPARSE and vector operations in cuBLAS/hipBLAS. Figure 8 illustrates the performance and speedups of our work compared to the baselines

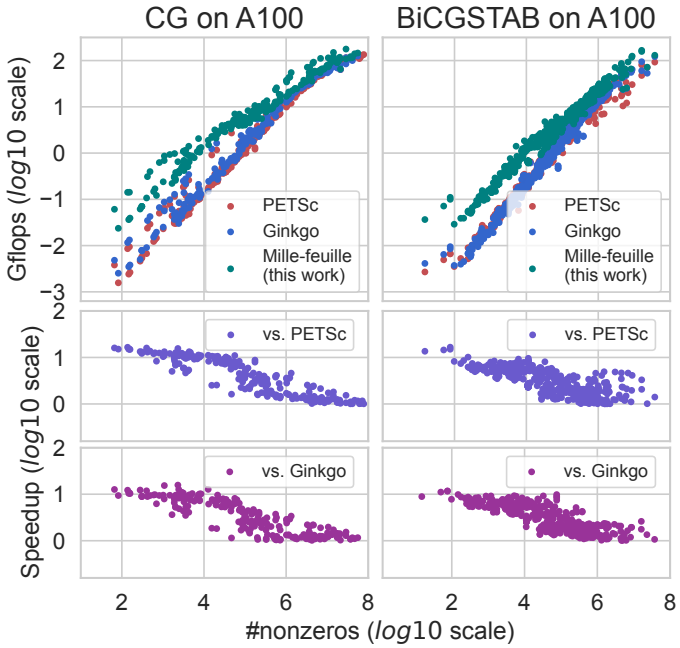


Fig. 9: Performance comparison of Mille-feuille over PETSc and Ginkgo on the A100 GPU with 100 iterations.

on A100 and MI210 GPUs, demonstrating that our method always shows the best performance for all tested matrices.

For the CG solver, our algorithm achieves an average geometric mean speedup of 3.03x and 2.68x (up to 8.77x and 7.14x) compared with cuSPARSE and hipSPARSE, respectively. Regarding the BiCGSTAB solver, our algorithm achieves an average geometric mean speedup of 2.65x and 2.32x (up to 7.51x and 6.63x) compared with the baseline.

The best speedups are observed in the matrices 'bcsstm22', 'Trec4', 'mhdb416' and 'b1_ss'. For these four matrices,

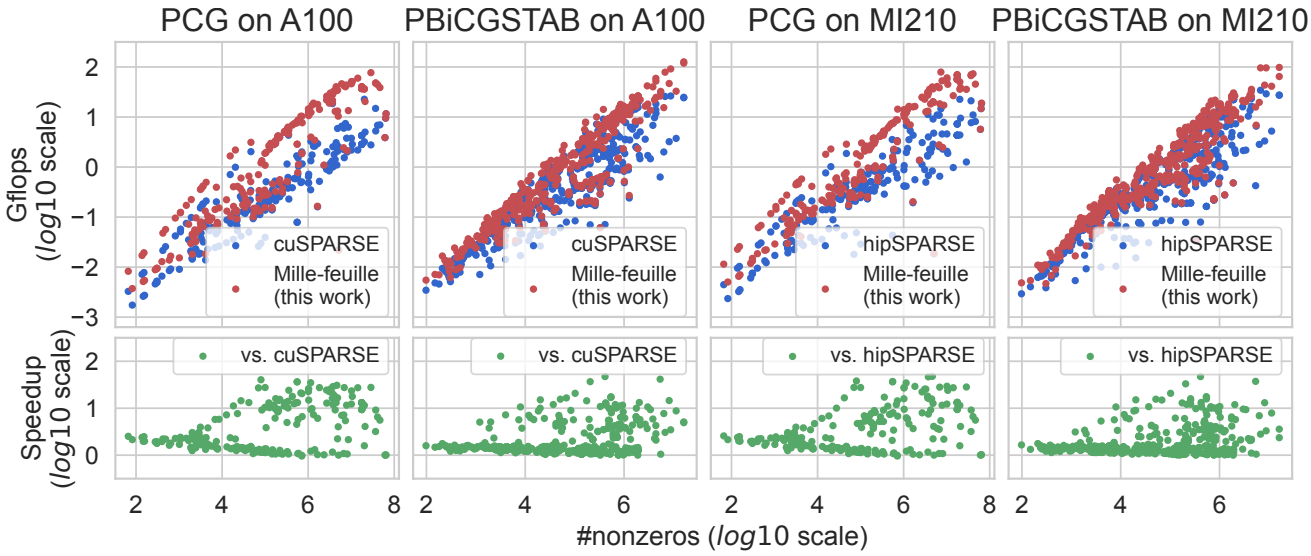


Fig. 10: Performance comparisons between Mille-feuille and the baseline implemented by calling cuSPARSE and hipSPARSE of preconditioned CG and BiCGSTAB on the NVIDIA A100 and AMD MI210 GPUs with 100 iterations, respectively.

synchronization constitutes over 50% of the total time in the baseline implementation using cuSPARSE and hipSPARSE. Our approach significantly reduces synchronization overhead between kernels by merging different kernels into one, thereby enhancing performance.

We then compare our Mille-feuille solver with two state-of-the-art linear solver libraries, PETSc and Ginkgo with 100 iterations on A100 GPU. As shown in Figure 9, our method also achieves the best performance for all tested matrices. Specifically, for the CG method, our algorithm achieves an average geometric mean speedup of 5.37x and 4.36x (up to 16.54x and 15.69x) compared with PETSc and Ginkgo. For the BiCGSTAB method, our algorithm achieves an average geometric mean speedup of 3.57x and 3.78x (up to 16.64x and 11.73x) compared with PETSc and Ginkgo. The best performance are observed in the matrices 'bcsstm22', 'mhdb416', 'jgl011', and 'rgg010', respectively. Similar to the previous observations, these matrices also exhibit significant synchronization overhead. Our algorithm circumvents these overheads to achieve high performance.

It is also observed that as the number of nonzeros in the matrix increases, the performance of our method exhibits a gradual decline (particularly when the number of nonzeros exceeds 10^6). This is primarily attributed to the trade-off between reducing synchronization costs through a single kernel and incurring additional overhead from atomic operations in scheduling. Consequently, in such scenarios, our solver opts to execute across multiple kernels.

C. Comparsion over Preconditioned Baseline

We compare the time taken for 100 iterations between Mille-feuille with the preconditioner and the baseline on CG and BiCGSTAB methods calling SpMV, SpTRSV in cuSPARSE/hipSPARSE and vector operations in cuBLAS/hipBLAS. Figure 10 illustrates the performance and speedups of our work compared to the baselines on A100 and MI210 GPUs, demonstrating that our method always shows the best performance for all tested matrices.

For the preconditioned CG solver, our algorithm achieves an average geometric mean speedup of 3.82x and 3.47x (up to 40.38x and 47.75x) compared with cuSPARSE and hipSPARSE, respectively. Regarding the preconditioned BiCGSTAB solver, our algorithm achieves an average geometric mean speedup of 1.79x and 1.63x (up to 45.63x and 44.34x) compared with the baseline.

The best speedups are observed in the matrices 'LFAT5000', 'ship_001', 'cz40948' and 'Chebyshev4'. For these matrices with high parallelism blocks, our recursive block SpTRSV algorithm divides them into more sub-matrices and computes square part using SpMV, which improves cache locality and parallelism, thereby enhancing performance.

D. Effectiveness of Mixed Precision

To further demonstrate the performance gains that mixed precision brings to our algorithm, we analyze the speedups of different precisions over only FP64 on 24 representative matrices with diverse precisions in CG and BiCGSTAB. As shown in Figure 11, we can find that our algorithm achieves different performances as the ratio of different precision varies.

Specifically, for most matrices, lower precision computations result in better performance, and different combinations of precision result in distinct performance behaviors. As can be seen, better performance is achieved while using more lower precisions (e.g., 'torso2') as opposed to a combination of low and high precisions (e.g., 't2dal_bci').

Furthermore, for matrices with high bypass rates, such as 'shallow_water1' and 'rajat24', Mille-feuille achieves the highest speedups because we remove the overhead of loading and computing the corresponding nonzeros. However, for the

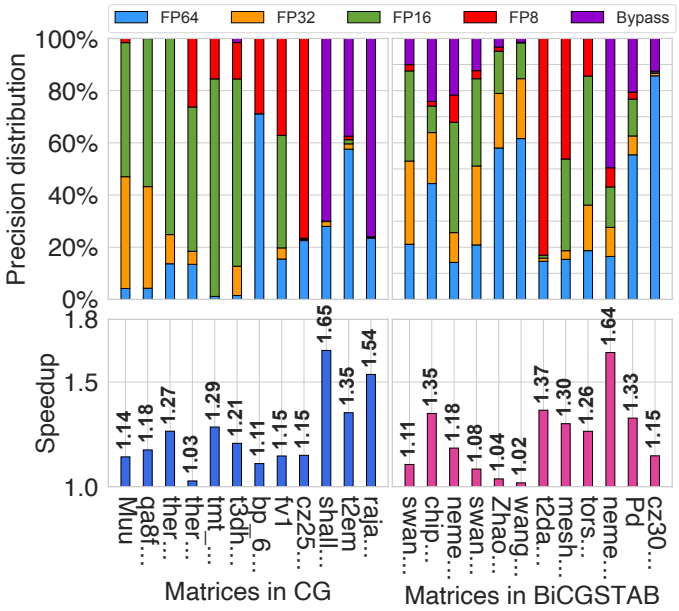


Fig. 11: The precision distribution across various tiles, and the performance gains of mixed precision in Mille-feuille.

matrix 'thermal' and 'wang1', although their low-precision ratios are high, they do not achieve higher speedups due to their small sizes (for such matrices, most speedups are already achieved from the single-kernel scheme).

E. Convergence Analysis

We further analyze 14 matrices (six from CG and eight from BiCGSTAB) that converge within 200 iterations. We first compare the number of iterations between our mixed precision version and the baseline implemented with FP64. As shown in Table II, for all matrices that converge, our mixed precision scheme often reasonably brings more iterations. It has an average of 1.06x (up to 1.47x) higher number of iterations compared to the baseline with cuSPARSE.

In addition, we analyze the variation of relative error between our mixed precision method and the benchmark algorithm with FP64 precision over the iterations of three matrices. As shown in Figure 12, for the matrix 'minsurfo', the trends of the relative errors of the two methods are comparable. For 'm3paltes', the relative error reduction rate of the mixed precision method is comparatively slower than that of the baseline. For 'poisson3Da', the relative error reduction rates of the two methods alternately increase and finally converge.

F. Correlation of the Number of Iterations and Runtime

For the 14 convergence matrices, we conduct a comparison between the solution time of Mille-feuille and the baseline time of the cuSPARSE implementation. As presented in Table II, across all matrices, our method consistently achieves shorter solution times, despite the higher or equal number of iterations required. For example, for the matrix 'mesh3e1', our method takes 1.47x more iterations (53 vs. 36), but it is 2.89x faster (1.19 ms vs. 3.44 ms) than the baseline. For 'pores_1' taking

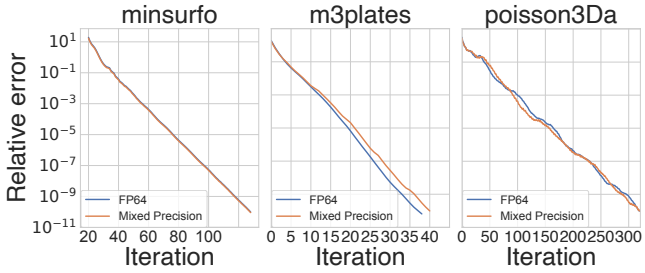


Fig. 12: The convergence comparison of relative error variation with iterations between mixed precision Mille-feuille and FP64 precision.

the same number of iterations (43), our method is 5.83x faster (1.25 ms vs. 7.29 ms). This can be attributed to the benefits derived both from single-kernel execution and from mixed precision.

Matrix	With cuSPARSE		Mille-feuille	
	#iter	time (ms)	#iter	time (ms)
CG				
mesh3e1	**36**	3.44	53	**1.19**
Muu	**63**	7.02	67	**4.91**
minsurfo	**109**	15.67	**109**	**15.34**
qa8fm	**69**	15.67	72	**4.91**
thermomech_TC	**86**	18.92	88	**15.03**
m3plates	**38**	4.54	40	**1.52**
BiCGSTAB				
CAG_mat72	**80**	14.87	87	**2.57**
arc130	**10**	1.71	11	**0.36**
fs_541_1	**7**	1.60	10	**0.43**
poli	**24**	**4.92**	31	**4.92**
Hamrle1	**112**	19.50	165	**4.79**
pores_1	**43**	7.29	**43**	**1.25**
cz308	**100**	17.44	147	**7.31**
majorbasis	**123**	52.91	140	**51.52**

TABLE II: The number of iterations and the solution time for matrices can converge in CG and BiCGSTAB with cuSPARSE and in Mille-feuille. The best numbers are highlighted.

G. Memory Cost Comparison

We compare the memory cost of our Mille-feuille with the standard three-array CSR in the cuSPARSE. As shown in Figure 13, our 2D tiled sparse data structure on average takes 1.04x more space than cuSPARSE. The main reason is that our two-level tiled structure typically needs to store more row indices, precision information, tile column indices, the number of nonzeros and non-empty rows in the tiles. Although the tiled format incurs additional arrays, incorporating different precisions for the initial values helps reduce the memory overhead of storing floating-point values in A. Nevertheless, according to the results, it is worth using the extra memory space because it makes our single-kernel and mixed precision strategy more convenient for faster execution.

H. Preprocessing Overhead Analysis

We further demonstrate the preprocessing overhead, including matrix format conversion, computational task distribution,

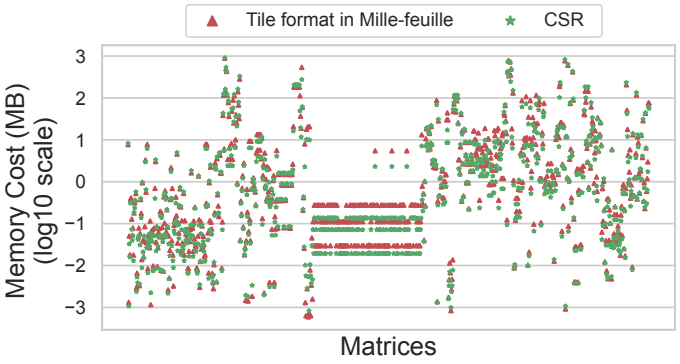


Fig. 13: Memory cost comparison of the tile format developed in Mille-feuille and the standard CSR format.

and initial precision assignment, of all matrices tested in the CG and BiCGSTAB methods. Figure 14 shows the cost breakdown of the preprocessing and 100 iterations of CG and BiCGSTAB. As can be seen, the preprocessing cost often does not exceed a single CG iteration, and takes a negligible proportion of the procedure of 100 iterations.

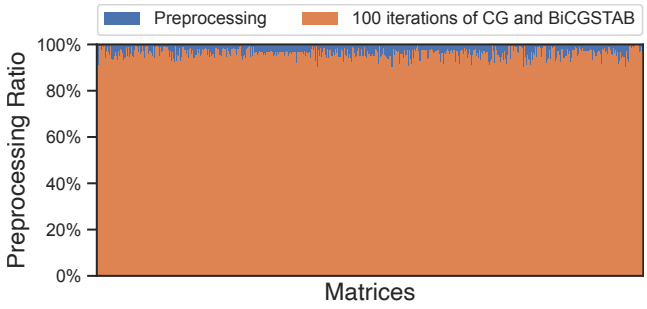


Fig. 14: Comparison of the proportion of preprocessing to the total runtime of CG and BiCGSTAB with 100 iterations of all tested matrices on A100.

V. Related Work

As probably the most widely used high performance iterative methods, **parallel Krylov subspace algorithms** have attracted much attention. Various optimization techniques, such as IDR algorithm [42], preconditioning methods [43]–[51], recycling technique [52], residual replacement strategy [53], block schemes [54]–[56], tensor method [57], recompute work [58], low energy approach [59], adaptive scheme [60], batched method [10], domain decomposition [61], robustness [62], [63], randomization [64]–[66], stability analysis [67], resistive random-access memory [68], [69], data transfer problems [4], [11], [70]–[72] and large-scale scalability [12], [73], [74], have been proposed for faster implementation of the Krylov subspace. A number of studies also considered the synchronization problem, and developed restarting [75], fine-grained policies [76]–[78], bulk method [79], loop optimization [80], kernel fusion [81], [82], synchronization-free [83]–[86] as well as several tradeoffs with data movement [87] techniques. Compared to those studies, our Mille-feuille is a new iterative solver considering both the use of mixed precision and the minimization of synchronization costs on modern GPUs.

With the strong demand from artificial intelligence computing, modern processors provide a large number of low precision compute units. This trend presents opportunities for **mixed precision assisted scientific computing**. Van Den Eshof et al. [88] and Clark et al. [89] demonstrated that Krylov subspace methods can benefit from multiple precision computations. Iterative refinement is one of the main scenarios for mixed precision [17], [21], [90], [91], and a sparse matrix can be stored with multiple precision [18]. The extended and mixed precision BLAS library [92], Ginkgo [34] and AmgT [31] provide various mixed precision kernels. Several studies [93]–[95] surveyed the use of mixed precision in numerical linear algebra methods. In contrast, the Mille-feuille proposed in this work supports tile-grained mixed precision, and works in the single kernel for dynamic precision conversion within the on-chip memory at runtime of the kernel.

Much research has focused on **speeding up SpMV**, typically the most time consuming kernel in iterative methods. Various techniques have been developed, such as locality aware [96]–[101], load balancing [9], [102]–[105], vector friendly [106]–[108], tensor core friendly [109], blocking [37], [110], [111], and reordering [112]. In particular, SpMV [18], [36], [113] can also be accelerated with mixed precision. In comparison, the SpMV in Mille-feuille obtains benefits from the tile-wise mixed precision data layout, and works with other components in a single kernel for higher performance.

VI. Conclusion

We have proposed the Mille-feuille solver with the characteristics of tile-grained mixed precision, a single kernel scheme with minimized synchronization costs, and dynamic precision conversion at runtime for partial convergence of the solution vector x. A comprehensive performance benchmark shows that the Mille-feuille solver obtained significant speedups on CG, BiCGSTAB, PCG and PBiCGSTAB over existing work on NVIDIA and AMD platforms.

Acknowledgments

We greatly appreciate the invaluable comments of all the reviewers. Zhou Jin is the corresponding author of this paper. This work was supported by the National Key R&D Program of China (Grant No. 2023YFB3001605), the National Natural Science Foundation of China (Grant No. 62204265, No. 62234010, No. 62372467, No. U23A20301, No. T2125013, No. 62032023 and No. 62102396), the State Key Laboratory of Computer Architecture (ICT, CAS) (Grant No. CARCHA202115) , the Beijing Nova Program (Grant No. Z211100002121143, No. 20220484217) and the Youth Innovation Promotion Association of Chinese Academy of Sciences (Grant No. 2021099). We are also very grateful to the State Key Laboratory of Processors for their support in the experimental hardware, and to Hemeng Wang for the helpful discussion.

REFERENCES

[1] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[2] H. A. Van der Vorst, *Iterative Krylov methods for large linear systems*. Cambridge University Press, 2003.

[3] M. Hoemmen, "Communication-avoiding krylov subspace methods," Ph.D. dissertation, University of California, Berkeley, USA, 2010.

[4] E. C. Carson, "Communication-avoiding krylov subspace methods in theory and practice," Ph.D. dissertation, University of California, Berkeley, USA, 2015.

[5] M. H. Gutknecht, "A brief introduction to krylov space methods for solving linear systems," in *Proceedings of the International Symposium on Frontiers of Computational Science*, 2007.

[6] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, 1952.

[7] H. A. van der Vorst, "Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 13, no. 2, 1992.

[8] I. Yamazaki, E. C. Carson, and B. Kelley, "Mixed precision s-step conjugate gradient with residual replacement on gpus," in *IPDPS*, 2022.

[9] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás, "Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units," *Concurr. Comput. Pract. Exp.*, vol. 34, no. 14, 2022.

[10] A. Kashi, P. Nayak, D. Kulkarni, A. Scheinberg, P. Lin, and H. Anzt, "Integrating batched sparse iterative solvers for the collision operator in fusion plasma simulations on gpus," *J. Parallel Distributed Comput.*, vol. 178, 2023.

[11] L. Grigori, S. Moufawad, and F. Nataf, "Enlarged krylov subspace conjugate gradient methods for reducing communication," *SIAM J. Matrix Anal. Appl.*, vol. 37, no. 2, 2016.

[12] L. Grigori and O. Tissot, "Scalable linear solvers based on enlarged krylov subspaces with dynamic reduction of search directions," *SIAM J. Sci. Comput.*, vol. 41, no. 5, 2019.

[13] M. Baboulin, S. Donfack, J. J. Dongarra, L. Grigori, A. Rémy, and S. Tomov, "A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines," in *ICCS*, 2012.

[14] C. Severance, "Ieee 754: An interview with william kahan," *Computer*, vol. 31, no. 3, 1998.

[15] D. G. Hough, "The ieee standard 754: One for the history books," *Computer*, vol. 52, no. 12, 2019.

[16] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, 1991.

[17] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, and A. Remón, "A mixed-precision algorithm for the solution of lyapunov equations on hybrid CPU-GPU platforms," *Parallel Comput.*, vol. 37, no. 8, 2011.

[18] K. Ahmad, H. Sundar, and M. Hall, "Data-driven mixed precision sparse matrix vector multiplication for gpus," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, 2019.

[19] E. C. Carson, N. J. Higham, and S. Pranesh, "Three-precision gmres-based iterative refinement for least squares problems," *SIAM J. Sci. Comput.*, vol. 42, no. 6, 2020.

[20] A. Abdelfattah, H. Anzt, E. G. Boman, E. C. Carson, T. Cojean, J. J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. A. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. J. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 4, 2021.

[21] E. Oktay and E. C. Carson, "Multistage mixed precision iterative refinement," *Numer. Linear Algebra Appl.*, vol. 29, no. 4, 2022.

[22] H. Anzt, P. Luszczek, J. J. Dongarra, and V. Heuveline, "Gpu-accelerated asynchronous error correction for mixed precision iterative refinement," in *Euro-Par*, 2012.

[23] X. Fu, B. Zhang, T. Wang, W. Li, Y. Lu, E. Yi, J. Zhao, X. Geng, F. Li, J. Zhang, Z. Jin, and W. Liu, "Pangulu: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems," in *SC*, 2023.

[24] T. Wang, W. Li, H. Pei, Y. Sun, Z. Jin, and W. Liu, "Accelerating sparse lu factorization with density-aware adaptive matrix multiplication for circuit simulation," in *DAC*, 2023.

[25] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou, "Sflu: Synchronization-free sparse lu factorization for fast circuit simulation on gpus," in *DAC*, 2021.

[26] Y. Lu, Y. Luo, H. Lian, Z. Jin, and W. Liu, "Implementing lu and cholesky factorizations on artificial intelligence accelerators," *CCF Trans. High Perform. Comput.*, vol. 3, no. 3, 2021.

[27] H. Anzt, J. J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers," *Concurr. Comput. Pract. Exp.*, vol. 31, no. 6, 2019.

[28] G. Flegar, H. Anzt, T. Cojean, and E. S. Quintana-Ortí, "Adaptive precision block-jacobi for high performance preconditioning in the ginkgo linear algebra software," *ACM Trans. Math. Softw.*, vol. 47, no. 2, 2021.

[29] E. C. Carson and N. Khan, "Mixed precision iterative refinement with sparse approximate inverse preconditioning," *SIAM J. Sci. Comput.*, vol. 45, no. 3, 2023.

[30] Y. M. Tsai, N. Beams, and H. Anzt, "Three-precision algebraic multigrid on gpus," *Future Gener. Comput. Syst.*, vol. 149, 2023.

[31] Y. Lu, L. Zeng, T. Wang, X. Fu, W. Li, H. Cheng, D. Yang, Z. Jin, M. Casas, and W. Liu, "Amgt: Algebraic multigrid solver on tensor cores," in *SC*, 2024.

[32] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.

[33] R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and J. Zhang, "Toward performance-portable PETSc for GPU-based exascale systems," *Parallel Comput.*, vol. 108, 2021.

[34] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," *ACM Trans. Math. Softw.*, vol. 48, no. 1, 2022.

[35] E. C. Carson, T. Gergelits, and I. Yamazaki, "Mixed precision s-step lanczos and conjugate gradient algorithms," *Numer. Linear Algebra Appl.*, vol. 29, no. 3, 2022.

[36] E. Tezcan, T. Torun, F. Koşar, K. Kaya, and D. Unat, "Mixed and multi-precision spmv for gpus with row-wise precision selection," in *SBAC-PAD*, 2022.

[37] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *IPDPS*, 2021.

[38] H. Ji, H. Song, S. Lu, Z. Jin, G. Tan, and W. Liu, "Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on gpus," in *ICPP*, 2022.

[39] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "Tilespgemm: A tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus," in *PPoPP*, 2022.

[40] Z. Lu and W. Liu, "Tilesptrsv: a tiled algorithm for parallel sparse triangular solve on gpus," *CCF Trans. High Perform. Comput.*, vol. 5, no. 2, 2023.

[41] Z. Lu, Y. Niu, and W. Liu, "Efficient block algorithms for parallel sparse triangular solve," in *ICPP*, 2020.

[42] H. Anzt, M. Kreutzer, E. Ponce, G. D. Peterson, G. Wellein, and J. J. Dongarra, "Optimization and performance evaluation of the IDR iterative krylov solver on gpus," *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 2, 2018.

[43] M. Manguoglu, M. Koyutürk, A. H. Sameh, and A. Grama, "Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers," *SIAM J. Sci. Comput.*, vol. 32, no. 3, 2010.

[44] D. Kressner, M. Plesinger, and C. Tobler, "A preconditioned low-rank CG method for parameter-dependent lyapunov matrix equations," *Numer. Linear Algebra Appl.*, vol. 21, no. 5, 2014.

[45] A. Mang and G. Biros, "A semi-lagrangian two-level preconditioned newton-krylov solver for constrained diffeomorphic image registration," *SIAM J. Sci. Comput.*, vol. 39, no. 6, 2017.

[46] H. Al Daas and L. Grigori, "A class of efficient locally constructed preconditioners based on coarse spaces," *SIAM J. Matrix Anal. Appl.*, vol. 40, no. 1, 2019.

[47] H. Al Daas, T. Rees, and J. Scott, "Two-level nyström–schur preconditioner for sparse symmetric positive definite matrices," *SIAM J. Sci. Comput.*, vol. 43, no. 6, 2021.

[48] V. Nguyen and L. Grigori, "Interpretation of parareal as a two-level additive schwarz in time preconditioner and its acceleration with GMRES," *Numer. Algorithms*, vol. 94, no. 1, 2023.

[49] H. Al Daas, P. Jolivet, and T. Rees, "Efficient algebraic two-level schwarz preconditioner for sparse matrices," *SIAM J. Sci. Comput.*, vol. 45, no. 3, 2023.

[50] L. Grigori, F. Nataf, and L. Qu, "Overlapping for preconditioners based on incomplete factorizations and nested arrow form," *Numer. Linear Algebra Appl.*, vol. 22, no. 1, 2015.

[51] Y. Zhao, X. Yang, Y. Bai, L. Zeng, D. Niu, W. Liu, and Z. Jin, "Csp: Comprehensively-sparsified preconditioner for efficient nonlinear circuit simulation," in *ICCAD*, 2024.

[52] H. A. Daas, L. Grigori, P. Hénon, and P. Ricoux, "Recycling krylov subspaces and truncating deflation subspaces for solving sequence of linear systems," *ACM Trans. Math. Softw.*, vol. 47, no. 2, 2021.

[53] E. C. Carson and J. Demmel, "A residual replacement strategy for improving the maximum attainable accuracy of s-step krylov subspace methods," *SIAM J. Matrix Anal. Appl.*, vol. 35, no. 1, 2014.

[54] D. Kressner, K. Lund, S. Massei, and D. Palitta, "Compress-and-restart block krylov subspace methods for sylvester matrix equations," *Numer. Linear Algebra Appl.*, vol. 28, no. 1, 2021.

[55] A. Frommer, K. Lund, and D. B. Szyld, "Block krylov subspace methods for functions of matrices ii: Modified block fom," *SIAM J. Matrix Anal. Appl.*, vol. 41, no. 2, 2020.

[56] D. Kressner, Y. Ma, and M. Shao, "A mixed precision lobpcg algorithm," *Numer. Algorithms*, vol. 94, no. 4, 2023.

[57] D. Kressner and C. Tobler, "Krylov subspace methods for linear systems with tensor product structure," *SIAM J. Matrix Anal. Appl.*, vol. 31, no. 4, 2010.

[58] T. Chen and E. C. Carson, "Predict-and-recompute conjugate gradient variants," *SIAM J. Sci. Comput.*, vol. 42, no. 5, 2020.

[59] J. I. Aliaga, J. Pérez, E. S. Quintana-Ortí, and H. Anzt, "Reformulated conjugate gradient for the energy-aware solution of linear systems on gpus," in *ICPP*, 2013.

[60] E. C. Carson, "The adaptive s-step conjugate gradient method," *SIAM J. Matrix Anal. Appl.*, vol. 39, no. 3, 2018.

[61] M. Manguoglu, "A domain-decomposing parallel sparse linear system solver," *J. Comput. Appl. Math.*, vol. 236, no. 3, 2011.

[62] M. Manguoglu and V. Mehrmann, "A robust iterative scheme for symmetric indefinite systems," *SIAM J. Sci. Comput.*, vol. 41, no. 3, 2019.

[63] H. Al Daas and P. Jolivet, "A robust algebraic multilevel domain decomposition preconditioner for sparse symmetric positive definite matrices," *SIAM J. Sci. Comput.*, vol. 44, no. 4, 2022.

[64] H. Al Daas, G. Ballard, P. Cazeaux, E. Hallman, A. Miedlar, M. Pasha, T. W. Reid, and A. K. Saibaba, "Randomized algorithms for rounding in the tensor-train format," *SIAM J. Sci. Comput.*, vol. 45, no. 1, 2023.

[65] O. Balabanov and L. Grigori, "Randomized gram-schmidt process with application to GMRES," *SIAM J. Sci. Comput.*, vol. 44, no. 3, 2022.

[66] A. Cortinovis, D. Kressner, and Y. Nakatsukasa, "Speeding up krylov subspace methods for computing $\boldsymbol{f(A)b}$ via randomization," *SIAM J. Matrix Anal. Appl.*, vol. 45, no. 1, 2024.

[67] E. C. Carson, M. Rozložník, Z. Strakos, P. Tichý, and M. Tuma, "The numerical stability analysis of pipelined conjugate gradient methods: Historical context and methodology," *SIAM J. Sci. Comput.*, vol. 40, no. 5, 2018.

[68] M. Fan, X. Chen, D. Yang, Z. Jin, and W. Liu, "Recg: Reram-accelerated sparse conjugate gradient," in *DAC*, 2024.

[69] M. Fan, X. Tian, Y. He, J. Li, Y. Duan, X. Hu, Y. Wang, Z. Jin, and W. Liu, "Amgr: Algebraic multigrid accelerated on reram," in *DAC*, 2023.

[70] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. J. Dongarra, "Improving the performance of CA-GMRES on multicores with multiple gpus," in *IPDPS*, 2014.

[71] H. Anzt, S. Tomov, P. Luszczek, W. B. Sawyer, and J. J. Dongarra, "Acceleration of gpu-based krylov solvers via data transfer reduction," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 3, 2015.

[72] I. Ismayilov, J. Baydamirli, D. Sağbili, M. Wahib, and D. Unat, "Multi-gpu communication schemes for iterative solvers: When cpus are not in charge," in *ICS*, 2023.

[73] G. Biros and O. Ghattas, "Parallel lagrange-newton-krylov-schur methods for pde-constrained optimization. part I: the krylov-schur solver," *SIAM J. Sci. Comput.*, vol. 27, no. 2, 2005.

[74] M. Manguoglu, A. H. Sameh, and O. Schenk, "PSPIKE: A parallel hybrid sparse linear system solver," in *Euro-Par*, 2009.

[75] K. Lund, "Adaptively restarted block krylov subspace methods with low-synchronization skeletons," *Numer. Algorithms*, vol. 93, no. 2, 2023.

[76] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout, "Sparse computation data dependence simplification for efficient compiler-generated inspectors," in *PLDI*, 2019.

[77] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Parsy: inspection and transformation of sparse matrix computations for parallelism," in *SC*, 2018.

[78] A. Jangda, S. Maleki, M. M. Dehnavi, M. Musuvathi, and O. Saarikivi, "A framework for fine-grained synchronization of dependent GPU kernels," in *CGO*, 2024.

[79] A. N. Yzelman and R. H. Bisseling, "An object-oriented bulk synchronous parallel library for multicore programming," *Concurr. Comput. Pract. Exp.*, vol. 24, no. 5, 2012.

[80] B. Zarebavani, K. Cheshmi, B. Liu, M. M. Strout, and M. M. Dehnavi, "Hdagg: Hybrid aggregation of loop-carried dependence iterations in sparse matrix computations," in *IPDPS*, 2022.

[81] K. Rupp, J. Weinbub, A. Jüngel, and T. Grasser, "Pipelined iterative solvers with kernel fusion for graphics processing units," *ACM Trans. Math. Softw.*, vol. 43, no. 2, 2016.

[82] J. I. Aliaga, J. Pérez, and E. S. Quintana-Ortí, "Systematic fusion of cuda kernels for iterative sparse linear system solvers," in *Euro-Par*, 2015.

[83] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," in *Euro-Par*, 2016.

[84] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 21, 2017.

[85] J. Su, F. Zhang, W. Liu, B. He, R. Wu, X. Du, and R. Wang, "Capellinisptrsv: a thread-level synchronization-free sparse triangular solve on gpus," in *ICPP*, 2020.

[86] F. Zhang, J. Su, W. Liu, B. He, R. Wu, X. Du, and R. Wang, "Yuenye-ungsptrsv: a thread-level and warp-level fusion synchronization-free sparse triangular solve," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 9, 2021.

[87] E. Solomonik, E. C. Carson, N. Knight, and J. Demmel, "Tradeoffs between synchronization, communication, and computation in parallel linear algebra computations," in *SPAA*, 2014.

[88] J. Van Den Eshof and G. L. Sleijpen, "Inexact krylov subspace methods for linear systems," *SIAM J. Matrix Anal. Appl.*, vol. 26, no. 1, 2004.

[89] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, "Solving lattice qcd systems of equations using mixed precision solvers on gpus," *Comput. Phys. Commun.*, vol. 181, no. 9, 2010.

[90] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, "Error bounds from extra-precise iterative refinement," *ACM Trans. Math. Softw.*, vol. 32, no. 2, 2006.

[91] J. Demmel, Y. Hida, E. J. Riedy, and X. S. Li, "Extra-precise iterative refinement for overdetermined least squares problems," *ACM Trans. Math. Softw.*, vol. 35, no. 4, 2009.

[92] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision blas," *ACM Trans. Math. Softw.*, vol. 28, no. 2, 2002.

[93] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 4, 2021.

[94] N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numer.*, vol. 31, 2022.

[95] X. Lei, T. Gu, S. Graillat, X. Xu, and J. Meng, "Comparison of reproducible parallel preconditioned bicgstab algorithm based on exblas and reproblas," in *HPC Asia*, 2023.

[96] A. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 1, 2014.

[97] A. N. Yzelman and R. H. Bisseling, "Two-dimensional cache-oblivious sparse matrix-vector multiplication," *Parallel Comput.*, vol. 37, no. 12, 2011.

[98] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *IPDPS*, 2011.
[99] K. Cheshmi, Z. Cetinic, and M. M. Dehnavi, "Vectorizing sparse matrix computations with partially-strided codelets," in *SC*, 2022.
[100] W. Li, H. Cheng, Z. Lu, Y. Lu, and W. Liu, "Haspmv: Heterogeneity-aware sparse matrix-vector multiplication on modern asymmetric multicore processors," in *CLUSTER*, 2023.
[101] X. Yu, H. Ma, Z. Qu, J. Fang, and W. Liu, "Numa-aware optimization of sparse matrix-vector multiplication on armv8-based many-core architectures," in *NPC*, 2020.
[102] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, 2020.
[103] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC*, 2016.
[104] H. Mi, X. Yu, X. Yu, S. Wu, and W. Liu, "Balancing computation and communication in distributed sparse matrix-vector multiplication," in *CCGrid*, 2023.
[105] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Comput.*, vol. 49, 2015.
[106] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, no. 2, 2013.
[107] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *ICS*, 2013.
[108] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *ICS*, 2015.
[109] Y. Lu and W. Liu, "DASP: specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication," in *SC*, 2023.
[110] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA*, 2009.
[111] E. Yi, Y. Duan, Y. Bai, K. Zhao, Z. Jin, and W. Liu, "Cuper: Customized dataflow and perceptual decoding for sparse matrix-vector multiplication on hbm-equipped fpgas," in *DATE*, 2024.
[112] J. D. Trotter, S. Ekmekçibaşı, J. Langguth, T. Torun, E. Düzakın, A. Ilic, and D. Unat, "Bringing order to sparsity: A sparse matrix reordering study on multicore cpus," in *SC*, 2023.
[113] K. Isupov, "Multiple-precision sparse matrix–vector multiplication on gpus," *J. Comput. Sci.*, vol. 61, 2022.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 We develop a tiled sparse format to store a sparse matrix in tiles with different initial precisions.

C_2 We leverage atomic operations that make the whole solving procedure work within a single GPU kernel.

C_3 We enable tile-wise on-chip dynamic precision conversion within the single kernel at runtime.

B. Computational Artifacts

https://doi.org/10.5281/zenodo.12589363

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact names Mille-feuille, an effective high performance CG and BiCGSTAB solver that (1) stores a sparse matrix in tiles with different initial precisions (C_1), (2) uses just one kernel call to complete SpMV, dot product, and AXPY in an entire procedure, instead of calling a number of individual CUDA kernels (C_2), and (3) changes the precision of a column of tiles in on-chip memory when the corresponding elements in the solution vector x are partially convergenced. (C_3) The experimental numbers demonstrate that our Mille-feuille outperforms cuSPARSE/hipSPARSE, PETSc, and Ginkgo by a factor of 3.03x/2.68x, 5.37x, 4.36x in CG, 2.65x/2.32x, 3.57x, 3.78x in BiCGSTAB, respectively.

Expected Results

This artifact contains Mille-feuille, cuSPARSE/hipSPARSE, PETSc, and Ginkgo. In the test results, the execution time of Mille-feuille is shorter than that of cuSPARSE/hipSPARSE, PETSc, and Ginkgo.

Expected Reproduction Time (in Minutes)

The estimated time to download the datasets is 1 hours.

The estimated time to compile this artifact is 1 hours.

The expected computational time of this artifact on NVIDIA A100 GPU is 6 hours.

Artifact Setup (incl. Inputs)

Hardware: GPU: NVIDIA A100 GPU (PCIe, 80GB, 1.94TB/s).

Disk Space: at least 60 GB (to store the experiment input dataset).

Software: Mpich v3.4.1 or above;
NVIDIA CUDA Toolkit v12.0;
cuSPARSE v12.0;
cuBLAS v12.0;
GCC v11.3.0 or above;
Python 3.9 or above;
cmake v3.26 or above;
Pandas package v2.2.2.
matplotlib package v3.9.0.
seaborn package v0.13.2

Datasets / Inputs: All 230 symmetric positive-definite matrices for CG and 686 nonsymmetric or indefinite matrices for BiCGSTAB, from the entire SuiteSparse Matrix Collection (https://sparse.tamu.edu/).

Installation and Deployment: Compiling on NVIDIA machine requires drivers for CUDA (nvcc v12.0). Additionally, gcc v11.3.0, cmake v3.16, and OpenMP are required.

Artifact Execution

Our workflow consists of four tasks: T_1, T_2, T_3, and T_4. Task T_1 downloads the CG and BiCGSTAB datasets from the SuiteSparse Matrix Collection using the dataset names in matrix_set.csv. In task T_2, we need to set up the compile environment and run script 'compile.sh', generating all executable files. The generated dataset serves as input for computational task T_3 via a script named test.sh, which runs all tested algorithms automatically. The output of T_3 is processed by task T_4 through the python script, producing the final result plot, Figure 8, Figure 9, Figure 10, Figure 11, Figure 12 and Figure 13. The tasks are dependent on each other in the following order: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$."

For our experiments, the input parameters include matrix names in the matrix market format (e.g., '*.mtx'). The dataset consists of 230 matrices for CG and 686 matrices for BiCGSTAB. The number of repetitions is 100.

Artifact Analysis (incl. Outputs)

By analyzing the output using the script test.sh, we obtained the average and maximum speedup of our Mille-feuille compared to other methods, as shown in Figures 8 and 9. The varying performances and precision ratios are presented in Figures 10 and 11. The memory cost of our method and cuSPARSE is illustrated in Figure 12. The cost breakdown of the preprocessing and 100 iterations of CG and BiCGSTAB is shown in Figure 13.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Installation and compilation (30 minutes): Download from the link (https://doi.org/10.5281/zenodo.12589363). Once downloaded locally, use the command to unzip it.

$unzip Mille-feuille.zip

$cd Mille-feuille

Mille-feuille requires NVCC version If the default compiler version of the current environment does not meet the requirements, users need to manually change the cuda install path in the Mille-feuille/Makefile file:

line1: CUDA_INSTALL_PATH = /Your_CUDA_path.

After that, in the artifact directory Mille-feuille/, compile the project.

$make

Preparation for Dataset (1 hours or more): Our experimental dataset includes all 230 symmetric positive-definite matrices for CG and 686 nonsymmetric or indefinite matrices for BiCGSTAB in the SuiteSparse Matrix Collection. These matrices need to be downloaded locally via the script matrix.py provided in the artifact. This process will take approximately 1 hours (the total size is estimated to be 22GB) or longer. If users accept the simplified dataset, we have also prepared a small-scale dataset of 570 matrices via the script matrix_simple.py. The download of the simplified dataset will take approximately 45 minutes.

Dataset download command:

$python3 matrix.py or **$python3 matrix_simple.py**

All matrices will be stored in the directory matrix/.

Preparation for Comparative methods (1 hours or more): In this paper, we compare our method with cuSPARSE (v12.0), PETSc (https://petsc.org/release/install/download), and Ginkgo (https: //github.com/ginkgo-project/ginkgo). The PETSc and the Ginkgo code have been included in the current artifact.

To compile the PETSc, you can use the compile command:

$unzip petsc-main.zip

$cd ./petsc-main/

$bash compile_PETSc.sh

To compile the Ginkgo, you can use the compile command:

$unzip ginkgo-develop.zip

$cd ./ginkgo-develop/

$bash compile_Ginkgo.sh (When installing Ginkgo, sudo privileges are required, which are already included in the script, so you just need to enter your password.)

Artifact Execution

After the dataset and the other methods are ready, in the directory Mille-feuille/, run this command:

$bash test_performance.sh (5 hours or more)

This script includes performance tests of Mille-feuille, cuSPARSE, PETSc and Ginkgo of the target dataset matrices, and will generate some result files.

Then run the next command:

$bash test_memory.sh (2 hours or more)

This script includes memory tests of Mille-feuille and cuSPARSE of the target dataset matrices and will generate a result file.

Finally run the command:

$bash test_preprocess.sh (1 hours or more)

This script includes preprocessing overhead tests of Mille-feuille of the target dataset matrices and will generate a result file.

We also provide an all_figures.sh to plot all the figures in our paper directly. You can run it with the command:

$bash all_figures.sh

and get Figure 8, Figure 9, Figure 10, Figure 11 and Figure12 in this paper.

Artifact Analysis (incl. Outputs)

The script test_performance.sh will generate some performance plots similar to Figure 8, Figure 9, Figure 10 and Figure 11 in the paper. These figures are related to C_1, C_2 and C_3 in this paper.

The script test_memory.sh will generate a memory cost plot similar to Figure 12 in the paper, which is related to C_1 and C_3 in this paper.

The script test_preprocess.sh will generate a preprocessing overhead plot similar to Figure 13 in the paper, which is related to C_1 and C_2 in this paper.

You can also run the script all_figures.sh to get all figures in this paper that are related to C_1, C_2, and C_3.