Sparsified Preconditioned Conjugate Gradient Solver on GPUs

Da Ma
McMaster University
Department of Electrical and Computer Engineering
Hamilton, Ontario, Canada
mad29@mcmaster.ca

Khalid Ahmad
University of Utah
School of Computing
Salt Lake City, Utah, USA
Khalid@cs.utah.edu

Kazem Cheshmi
McMaster University
Department of Electrical and Computer Engineering
Hamilton, Ontario, Canada
cheshmi@mcmaster.ca

Hari Sundar
University of Utah
School of Computing
Salt Lake City, Utah, USA
hari@cs.utah.edu

Mary Hall
University of Utah
School of Computing
Salt Lake City, Utah, USA
mhall@cs.utah.edu

ABSTRACT

Preconditioned iterative sparse linear solvers are memory-efficient for large scientific simulations, but the dependences between iterations introduced by preconditioners limit parallelization. This issue is exacerbated on GPUs, which feature many parallel cores. We propose a sparsified preconditioned conjugate gradient (SPCG) solver that increases parallelism by reducing dependences through sparsification, while preserving convergence behavior. We evaluate the proposed SPCG using both ILU(0) and ILU(K) preconditioners on a wide range of symmetric positive definite (SPD) matrices. The proposed SPCG improves the performance of the iterative phase of SPCG by a geometric mean speedup of 1.23× and 1.65× over the non-sparsified PCG using ILU(0) and ILU(K), respectively on an NVIDIA A100 GPU. SPCG also yields geometric mean end-to-end speedups of 1.68× and 3.73× over the non-sparsified versions with ILU(0) and ILU(K), respectively, on the same platform.

ACM Reference Format:
Da Ma, Khalid Ahmad, Kazem Cheshmi, Hari Sundar, and Mary Hall. 2025. Sparsified Preconditioned Conjugate Gradient Solver on GPUs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The solution to sparse linear systems represents a critical computation in a large number of scientific simulation domains. Sparse linear systems are represented mathematically in Equation 1, where matrix A is sparse, vector b is given, and vector x is unknown. This work assumes A is symmetric positive definite (SPD).

$$Ax = b, \qquad x, b \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n} \tag{1}$$

Two distinct approaches have arisen for solving sparse linear systems. Direct solvers are more robust, but do not scale well due to large memory requirements. Iterative solvers, while less robust and sensitive to conditioning, have the advantage of being more scalable; because their memory usage remain invariant to the matrix structure. A common method to improve upon the robustness of iterative solvers is through preconditioning the solution matrix. Depending on the preconditioning choice, solver scalability and convergence are also affected. In this paper, we focus on a preconditioned conjugate gradient (PCG). PCG uses a transformed version of the $n \times n$ SPD linear system to improve convergence and solve the linear system faster.

This paper focuses on a class of preconditioners that rely on incomplete-LU factorization, deriving lower- and upper-triangular factors of the matrix. ILU(0) has no fill-in, which means that no additional nonzero elements are added after the factorization of the preconditioner. We also consider ILU(K), which introduces fill-in to produce the preconditioner's factors. ILU(K) offers more stability at increased complexity due to fill-in.

While preconditioning improves convergence of the solver, the construction and applications of the preconditioner can itself be costly. In dense ILU, the calculation for each row depends on the previous one, making it a sequential computation. When sparse matrices are used, not all columns have values, which makes it possible to use *wavefront parallelism* to perform calculations in parallel. In wavefront parallelism, independent iterations form a wavefront and a synchronization mechanism is required after each wavefront. Prior work [34, 36, 41] has used wavefront parallelism successfully; however, a large number of synchronizations can under-utilize GPU resources.

In this paper, we *sparsify* the linear system, eliminating nonzero values that are less likely to impact convergence. We call this approach Sparsified PCG or *SPCG*. Through sparsification, the preconditioning performs less computation and reduces data movement. But more importantly, we may increase parallelism by reducing the number of wavefront levels. Our findings suggest that SPCG can outperform regular PCG with better convergence, and the paper provides guidance on when SPCG can be effective and profitable.

The contributions of this paper are as follows:

(1) We present SPCG that uses a new wavefront-aware sparsification approach for iterative solvers that improves performance by reducing computation and data movement, and increasing parallelism.
(2) We provide a detailed analysis of when SPCG is profitable, based on matrix numerical characteristics and wavefront reduction. Analysis evaluates the correlation between wavefront reduction and performance improvement, portability

Conference'17, July 2017, Washington, DC, USA
2025. ACM ISBN 000-0-0000-0000-0/00/00…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Algorithm 1: Preconditioned Conjugate Gradient (PCG)

Input : Matrix A, preconditioner M, right-hand side b, tolerance ϵ, maximum iteration $k_{\max}$
Output: Solution x

1 $x_0 \leftarrow$ initial guess (e.g., zero)
2 $r_0 \leftarrow b - Ax_0$
3 $z_0 \leftarrow M^{-1} r_0$
4 $p_0 \leftarrow z_0$
5 **for** $k \leftarrow 0$ **to** $k_{\max}$ **do**
6 **if** $\|r_k\| < \epsilon$ **then**
7 **return** x_k
8 **end**
9 $w_k \leftarrow Ap_k$
10 $\alpha_k \leftarrow \frac{(r_k, z_k)}{(p_k, w_k)}$
11 $x_{k+1} \leftarrow x_k + \alpha_k p_k$
12 $r_{k+1} \leftarrow r_k - \alpha_k w_k$
13 $z_{k+1} \leftarrow M^{-1} r_{k+1}$
14 $\beta_k \leftarrow \frac{(r_{k+1}, z_{k+1})}{(r_k, z_k)}$
15 $p_{k+1} \leftarrow z_{k+1} + \beta_k p_k$
16 **end**
17 **return** $x_{k_{\max}}$

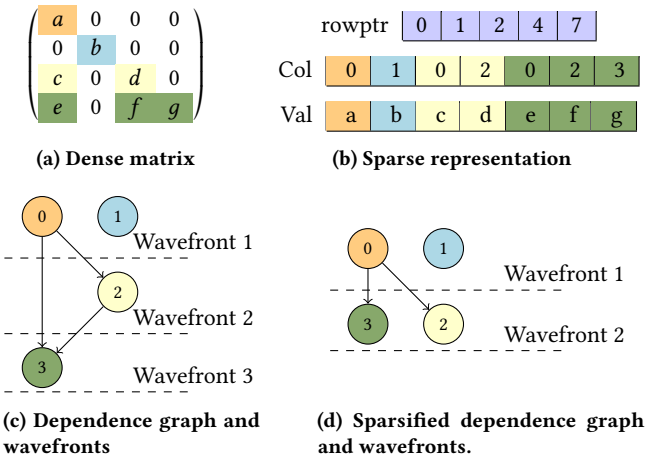
across two GPUs and one CPU, and examines how application domain impacts convergence.

(3) We present an implementation of the two solver variants for GPUs that provides a gmean end-to-end speedup of 1.68× and 3.73× over the original solvers with ILU(0) and ILU(K) preconditioners on A100, respectively.

2 MOTIVATION

The PCG method is a well-known iterative solver for sparse linear systems, especially in applications that exhibit SPD properties. A variant of the PCG algorithm, known as left-preconditioned CG [40], is described in Algorithm 1. As shown, PCG starts with an initial guess and then calculates the residual vector r, where A is the coefficient sparse matrix and b is the right-hand side vector. For a number of iterations $j = 0, 1, 2, \ldots$, the algorithm computes the step size using a preconditioner and calculates and updates the solution and residual vectors until the convergence condition is satisfied or the maximum number of iterations is reached. Several other algorithms can operate on SPD matrices, but PCG is usually the quickest and most reliable at solving those types of systems [6], hence the method of choice here.

In PCG, the computational cost is dominated by the sparse matrix-vector multiplication (SpMV) and the application of the preconditioner, as shown in Lines 9, 15, and 13 of Algorithm 1. While SpMV is inherently parallel, the primary challenge lies in the efficient parallelization of the preconditioner application (Line 13), which involves solving sparse triangular systems. Specifically, applying the preconditioner entails solving $LUz_{k+1} = r_{k+1}$, where L and U are sparse lower and upper triangular matrices. This operation requires a forward substitution followed by a backward substitution, both of which exhibit data dependences that limit fine-grained parallelism. Thus, efficient wavefront parallelism is critical for accelerating this stage.



(a) Dense matrix **(b) Sparse representation**

(c) Dependence graph and wavefronts **(d) Sparsified dependence graph and wavefronts.**

Figure 1: The effect of sparsification on reducing dependences and wavefront in preconditioned sparse solvers. a) a sparse lower triangular matrix L with 7 nonzeros stored in dense format. b) compressed sparse row (CSR) presentations of L shown in part 1a. c) dependence graph wavefront parallelism for solving $Lx = b$, dashed lines represent barrier synchronizations. d) dependence graph and wavefronts when nnz f is sparsified.

Wavefront parallelization is a well known technique for exploiting parallelism for loops with data dependences. In wavefront parallelization for sparse solvers, an inspector generates a graph of sparse dependences at runtime and uses it to identify independent iteration sets, i.e., wavefronts. Iterations within each wave can be executed in parallel. The executor portion of the code executes the waves sequentially due to the sparse dependences between iterations in different wavefronts. Figure 1a and 1b show a dense lower triangular factor and its corresponding compressed sparse row format (CSR), respectively. The dependence DAG of sparse lower triangular solver involving the sparse matrix in Figure 1b is shown in Figure 1c. Each vertex i represents operations corresponding to finding a solution of equation i (or x_i). The dotted lines in Figure 1c show three wavefronts of the the lower triangular system. Elements a and b can be processed in the same wavefront to compute x_0 and x_1 since columns 0 and 1 have no overlapping row. However, processing elements c, d, e, f, g to compute x_2 and x_3 must wait until row 0 is completed. In Figure 1d, the nonzero value f, corresponding to the edge between graph nodes 2 and 3, was eliminated or sparsified, thereby reducing the wavefront levels from three to two.

This example motivates the opportunity of selectively removing nonzeros. Not only does it reduce the size of the sparse matrix and associated computations, but it also has the potential to increase wavefront parallelism when preconditioning is applied. The next section describes the SPCG sparsification approach.

3 SPARSIFIED PRECONDITIONED CONJUGATE GRADIENT (SPCG) SOLVER

Sparsified Preconditioned Conjugate Gradient (SPCG) accelerates the PCG solver by selectively zeroing out data in the sparse matrix, i.e., sparsifying. This section first provides an overview of SPCG and then introduces the wavefront-based sparsification and the theory behind it. Additionally, we explain the two variants based on ILU(0) and ILU(K), referred to as SPCG-ILU(0) and SPCG-ILU(K), respectively.

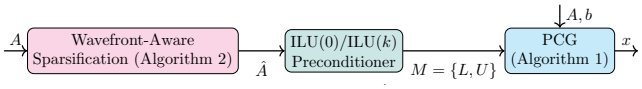


Figure 2: The SPCG solver overview

3.1 Overview

The overall view of SPCG is shown in Figure 2, where our proposed sparsification approach is highlighted. As shown, the SPCG solver takes a sparse SPD matrix A and a known right-hand side vector b as its inputs and computes the solution vector x. The SPCG solver first sparsifies A, then computes a preconditioner's factors using ILU(0)/ILU(K), followed by calling the PCG algorithm shown in Algorithm 1. To increase parallelism, SPCG uses a wavefront-aware sparsification approach to determine whether to eliminate nonzero values. An ILU solver is applied to sparsified A to compute preconditioner M, which has approximate L and U factors of A. The preconditioner, along with A and b, are used in the PCG algorithm to compute solution x. SPCG aims to use sparsification to run PCG efficiently and converge quickly.

3.2 Wavefront-Aware Sparsification

SPCG employs a wavefront-aware sparsification approach to selectively remove nonzero values from the original matrix in order to improve PCG convergence time. This process adds controlled perturbations to the matrix structure, with the goal of accelerating convergence while preserving numerical stability. Specifically, it seeks to balance two objectives: maintaining a bounded approximation error and reducing inter-iteration dependences of the preconditioner by lowering the number of parallel wavefronts. The following analysis focuses first on the convergence conditions under the injected sparsification error, followed by a discussion of algorithm and and the heuristics used for efficient implementation.

3.2.1 Theoretical Analysis. The goal of sparsification is to determine $\hat{A}$ in $A = \hat{A} + S$ where all three matrices are symmetric. We aim to show when convergence is guaranteed in solving $Ax = b$ under the introduced sparsification S.

To get the iterations after sparsification, we start from:

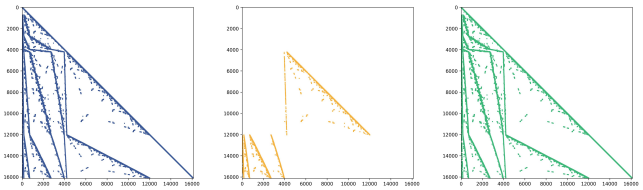
$$\hat{A}x + Sx = b \tag{2}$$

Treating x as the fixed point of an iterative scheme define:

$$\hat{A}x_{k+1} + Sx_k = b \tag{3}$$

Subtracting Equation 2 from Equation 3, we obtain:

$$\hat{A}(x_{k+1} - x) + S(x_k - x) = 0 \rightarrow x_{k+1} - x = \hat{A}^{-1}S(x - x_k) \tag{4}$$



(a) A **(b)** S **(c)** $\hat{A}$

Figure 3: Example patterns of matrix A, residual matrix S, and sparsified matrix $\hat{A}$. Matrix *Dubcova1* with 134,569 nonzero elements is sparsified with 10% and result in dropping 10.00% of nonzeros and 14.73% of wavefronts.

After taking the norm of both sides:

$$||x_{k+1} - x|| \leq ||\hat{A}^{-1}S|| \cdot ||x - x_k|| \tag{5}$$

Thus, if $||\hat{A}^{-1}S|| < 1$, the system is guaranteed to converge after applying sparsification. We simplify this condition by using the matrix norm sub-multiplicative property, i.e., the norm of a product is less than the product of norms:

$$||\hat{A}^{-1}S|| < ||\hat{A}^{-1}|| \cdot ||S|| \leq 1, \tag{6}$$

convergence is ensured if the product $||\hat{A}^{-1}|| \cdot ||S||$ is less than 1. Since $||\hat{A}^{-1}|| \cdot ||S||$ sets a looser bound than $||\hat{A}^{-1}S||$, even a value larger than 1 may also reflect an acceptable convergence safety.

We remark that in the theoretical derivation above, the term S implicitly includes not only the sparsification error but also the error introduced by the incomplete factorization, denoted as E_f. However, in practice, E_f is not available or easily computable. Therefore, in our algorithm, we reinterpret the condition $||\hat{A}^{-1}|| \cdot ||S|| < 1$ by isolating S to represent only the sparsification error, excluding E_f, and compare it against a relaxed empirical threshold τ instead of 1. This adjustment conceptually preserves the original convergence bound while enabling a practical and efficient implementation.

Computing $||\hat{A}^{-1}||$ after each sparsification step is computationally expensive. The cost of computing wavefronts must also be considered. To avoid this, we propose heuristics to approximately ensure the above condition with low computational overhead.

3.2.2 Algorithm. The wavefront-aware sparsification procedure is outlined in Algorithm 2, which takes as input the matrix A along with two thresholds: τ and ω to ensure convergence rate based on numerical properties and effective wavefront reduction. The algorithm is an iterative method that in each iteration computes $\hat{A}_t$, and then verifies that $\hat{A}_t$ satisfies both the convergence condition and the desired wavefront reduction. More specifically, convergence safety is assessed using Equation 6. The algorithm however uses certain approximation to achieve computational feasibility.

To avoid computing convergence and sparsity indicators on a per-nonzero basis, we predefine a set of three sparsification ratios t—10%, 5%, and 1%—each representing the percentage of nonzero entries to be removed from A. These ratios are computed experimentally and applied in decreasing order of aggressiveness as shown in line 2 in Algorithm 2. We observed that extending beyond these three sparsification ratios introduces diminishing returns. For each ratio t, as shown in line 3, a corresponding proportion of

Algorithm 2: Wavefront-Aware Sparsification

```
Input  : Matrix A, convergence threshold τ, wavefront threshold ω
Output : Matrix Â
/* Estimate convergence sensitivity */
1  w_A ← number of wavefronts in A
2  foreach t ∈ {10, 5, 1} do
       /* Check convergence indicator */
3      Â_t ← A − S_t
4      ‖Â_t^{-1}‖ ← κ(Â_t)/‖Â_t‖_2
5      if ‖Â_t^{-1}‖ · ‖S_t‖ > τ then
6          if t = 1 then return A − S_10
7          else continue with next t
8      end
       /* Check wavefront reduction */
9      w_{Â_t} ← number of wavefronts in Â_t
10     if 100(w_A − w_{Â_t})/w_{Â_t} ≥ ω or t = 1 then
11         return Â_t
12     end
13 end
14 return A − S_10
```

the smallest-absolute-magnitude nonzero entries is removed, while diagonal entries are preserved to maintain numerical stability. As a result, the original matrix A is decomposed into a sparsified matrix $\hat{A}_t$ and a residual matrix S_t containing the removed entries, such that $A = \hat{A}_t + S_t$. Figure 3 illustrates this sparsification and decomposition on a matrix from the SuiteSparse repository [18], where the effect is particularly visible in the lower triangular region.

For convergence testing, the algorithm uses an approximation for $\|\hat{A}^{-1}\|$ in Equation 6. The approximation is based on the general identity $k(\hat{A}) \approx \|\hat{A}^{-1}\| \cdot \|\hat{A}\|_2$ where $k(\hat{A})$ is the condition number of A and expensive to compute. Thus, the condition number $k(\hat{A})$ is approximated as the ratio of the inf-norm of $\hat{A}$, used as a proxy for its largest eigenvalue, to the smallest diagonal entry in $\hat{A}$, used as an approximation for the smallest eigenvalue. This estimation avoids expensive eigenvalue or inverse computations while still providing a practical indicator of convergence behavior. As shown in lines 3–8, for each sparsification ratio, the algorithm computes this convergence indicator $\|\hat{A}^{-1}\| \cdot \|S\|$ and compares against the predefined threshold τ. If this product exceeds τ, the algorithm proceeds to evaluate a smaller, more conservative ratio. However, if even the smallest ratio (1%) fails this convergence check, the most aggressive ratio (10%) is selected as shown in line 6 in Algorithm 2. Since no sparsification level guarantees reliable convergence, the algorithm instead prioritizes higher sparsification, potentially higher per-iteration speedup.

For sparsified matrices that satisfy the convergence test, the algorithm proceeds to evaluate wavefront reduction effectiveness, as shown in lines 9–12 of Algorithm 2. The number of wavefronts in the sparsified matrix based on t, denoted by $w_{\hat{A}_t}$, is computed and normalized against the wavefronts in the original matrix w_A to measure the relative reduction as shown in Equation 7:

$$\text{Wavefront reduction (\%)} = \frac{w_A - w_{\hat{A}}}{w_A} \times 100\%. \tag{7}$$

If the wavefront reduction exceeds the predefined threshold ω, the current sparsified matrix is considered effective, and the corresponding sparsified matrix $\hat{A}_t$ is selected as shown in line 11 in Algorithm 2. Otherwise, the algorithm proceeds to evaluate the next value of t. If none of the tested sparsification ratios achieves sufficient wavefront reduction despite satisfying the convergence criterion, suggesting limited potential for per-iteration acceleration, the algorithm selects the smallest t (1%), to minimize sparsification error, potentially reducing PCG iterations.

3.2.3 Heuristic choice analysis. The design of the wavefront-aware sparsification algorithm (Algorithm 2) incorporates a set of experimentally driven decisions to reduce computational overhead while maintaining robustness. Two key simplifications, restricting the sparsification ratio space and approximating condition number estimation, are evaluated and justified below based on extensive experiments on 107 SPD matrices from the SuiteSparse repository.

Algorithm 2 uses three empirically predetermined thresholds to balance convergence and sparsification overhead. The used sparsification ratios to the most consistently effective choices: {1, 5, 10}. To show smaller or larger thresholds provide limited advantage, we evaluated a broader range including 0.5, 15, 20, and 50. Results show that 0.5 brings negligible structural change; 86.92% cases have less than 5% relative wavefront reduction. Among them, 59.82% of matrices have no wavefronts reduced, limiting the benefit only from reduced FLOPs and minor parallelism gains. Conversely, while higher ratios such as 20 or 50 can yield stronger per-iteration speedups, they often lead to more degraded convergence. Specifically, 62.62% of matrices failed to converge or required at least twice the number of iterations at a ratio of 50. Nonetheless, as Algorithm 2 supports an extended set of candidate ratios, larger or smaller ratios may still be appropriate in practice when numerical or sparsity characteristics of matrices are known beforehand.

As discussed in Section 3.2.2, the algorithm uses an approximation of inverse norm and condition number to ensure convergence safety condition in Equation 6. The key approximation is in computing the condition number of $\hat{A}$ as the ratio of the inf-norm of $\hat{A}$ to its smallest diagonal entry. This inexpensive approximation is empirically working across the covered dataset. To validate this approximation, we computed the inverse norm with exact condition numbers using the same grid-searched optimal threshold combination ($\tau = 1$, $\omega = 10\%$). The approximated strategy achieved a geometric mean speedup of 1.233 with a convergence rate of 52.34%, while using the exact condition number gave 1.235 and 53.28%, respectively. These close results confirm that the approximation is accurate enough for guiding sparsification decisions.

3.3 SPCG-ILU(0) and -ILU(K)

The sparsified matrix $\hat{A}$ is then used to compute the preconditioner's factors as shown in Figure 2. In practice, a variety of preconditioning techniques could be used to improve the convergence and computational efficiency of an iterative method like PCG. In this paper, we focus on incomplete-LU(0) and ILU(K) preconditioning, resulting in SPCG-ILU(0) and SPCG-ILU(K) variants, respectively. ILU(0) factorization approximately decomposes a sparse matrix A into a lower triangular matrix L and an upper triangular matrix U, such that the non-zero sparsity structure of both triangular matrices matches

that of A without introducing any additional non-zero elements (fill-ins) during the factorization. This characteristic ensures that memory usage and computational costs remain low.

ILU(K) provides a more accurate preconditioner by introducing fill-ins during the computation of the factor. The resulting factor is denser than the ILU(0) factors but is still considered sparse. K in ILU(K) controls the number of fill-ins in the factor. As K increases, the number of fill-ins also increases. However, more accurate ILU(K) preconditioners require more memory, reaching a point where the trade-off between cost and accuracy becomes unfavorable because the runtime of the solver increases, even though the total number of iterations decreases.

We implemented the two variants of SPCG: SPCG-ILU(0) and SPCG-ILU(K). The SPCG-ILU(0) relies on existing Sparse ILU(0) preconditioners, such as cuSPARSE ILU(0) [36]. However, since dependence between iterations in ILU(K) changes during factorization, direct implementation in CUDA is significantly more challenging. Due to this technical difficulty and because the main focus of this work is on improving the solve phase of PCG (i.e., Algorithm 1), we use a CPU implementation of ILU(K) based on SuperLU [31] to compute the factors.

The induced sparsification exhibits a different effect on sparsity pattern of SPCG-ILU(0) and SPCG-ILU(K) variants. Since ILU(0) does not change the sparsity pattern of the input matrix, sparsification always reduces the number of nonzeros in the factor. Given the more complex interplay between sparsification and fill-in in ILU(K), the outcome is less predictable, as it remains unclear whether reducing non-zero elements will amplify, preserve, or diminish the gains provided by fill-in. In SPCG-ILU(K), since the goal is to evaluate the sparsification on the ILU(K) precondtioner, the value K is selected to be the same for both SPCG and PCG. Deciding K is often based on domain-specific information. To isolate the effect of K on sparsification and for fairness, we select the best converging K from 10, 20, 30, and 40 for a given matrix for the non-sparsified PCG-ILU(K). We then use this value to measure the effect of sparsification.

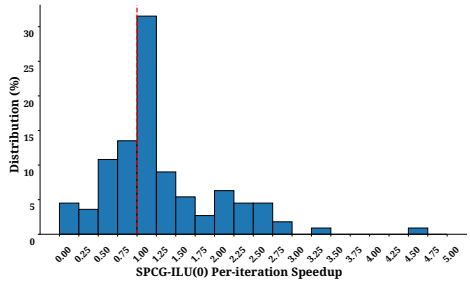
4 PERFORMANCE RESULTS

This section compares the performance of SPCG and the (non-sparsified) PCG and analyzes the effect of sparsification on SPCG's performance. Specifically, the following questions will be addressed: (1) How much faster is SPCG compared to the PCG solver? (2) How does sparsification affect the end-to-end performance of a linear solver? (3) How do different components of SPCG contribute to its performance? (4) How does SPCG perform across different GPU and CPU architectures?

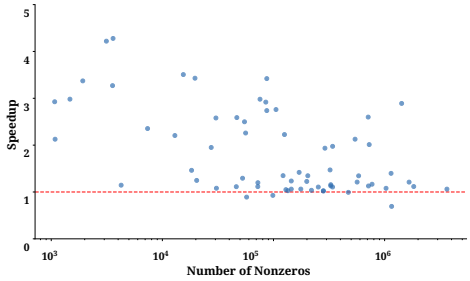
4.1 Setup

Matrix Dataset. We initially selected all SPD matrices from the SuiteSparse matrix repository [18] with dimensions greater than 1000. Matrices that produced NaN residuals under any configuration of fill factors during the iterative solving phase were excluded. This left a final set of 107 matrices, each with a complete set of results for both SPCG-ILU(0)/PCG-ILU(0) and SPCG-ILU(K)/PCG-ILU(K).

Architecture. Performance measurements are conducted on two NVIDIA GPU architectures, A100 and V100 and an AMD EPYC

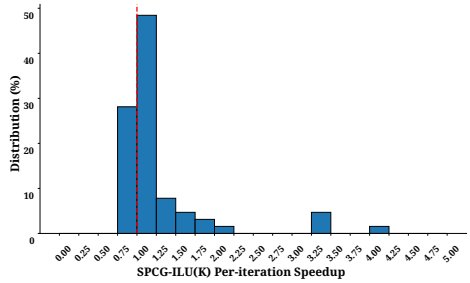


(a) Per-iteration speedups for SPCG-ILU(0).

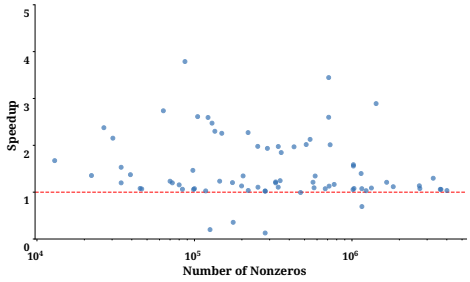


(b) The end-to-end speedups for SPCG-ILU(0).

Figure 4: SPCG-ILU(0) speedups on A100. The y-axis is limited to the speedups ranging from 0 to 5.



(a) Per-iteration speedups for SPCG-ILU(K).



(b) The end-to-end speedups of SPCG-ILU(K).

Figure 5: SPCG-ILU(K) speedups on A100. The y-axis is limited to the speedups ranging from 0 to 5.

7413 CPU architecture with 40 cores, operating at a base frequency of 2.65 GHz and capable of boosting up to 3.6 GHz.

Methods. We use the sparse ILU(0) and sparse CG solver from cuSPARSE as baselines in both SPCG-ILU(0) and SPCG-ILU(K). For computing GFLOP/sec, we compute the theoretical FLOPs of the non-sparsified baseline and reuse it for all methods. The reported measurements for SPCG-ILU(0) and SPCG-ILU(K) are based on the implemented heuristics that work by reducing the number of wavefronts and considering the matrix properties. The convergence threshold τ of 1 and wavefront threshold ω of 10% are selected based on a grid search over a swept range. All methods are implemented using single-precision.

4.2 Per-iteration Performance

Figure 4a shows the speedup distribution when the matrices are applied with SPCG-ILU(0). SPCG-ILU(0) provides a geometric mean speedup of 1.23× across all matrices in the dataset. Also, the SPCG-ILU(0) provides speedup for 69.16% of the matrices compared to the PCG-ILU(0). As shown, the SPCG-ILU(0) achieves speedup for the majority of matrices included in the dataset, while most of them are distributed within the range of 1 to 2. The range of GFLOP/sec for the original PCG baseline in Figure 4 is between 0.0004–156.2739 GFLOP/sec for ILU0.

Figure 5a demonstrates a similar trend for SPCG-ILU(K), where sparsification improves per-iteration performance for most matrices. The range of GFLOP/sec for the original PCG baseline in Figure 5 is between 0.0007–2.7090 GFLOP/sec for ILU(K). A notable point different from what was observed for SPCG-ILU(0) is that even when applying SPCG-ILU(K) leads to a slowdown, the slowdown still remains close to 1, suggesting a minor negative effect of SPCG-ILU(K) as compared to SPCG-ILU(0). SPCG-ILU(K) provides per-iteration speedup on 80.38% of matrices. The gmean speedup of SPCG-ILU(K) is 1.65× which is higher than SPCG-ILU(0). This can be attributed to the same reasoning as why sparsification tends to improve performance more on denser matrices: in ILU(K), more nonzeros are introduced during the factorization phase due to fill-ins; thus, more dependences are incurred. Sparsifying the matrix before ILU(K) factorization results in a sparser matrix input, potentially leading to more efficient parallel computation.

4.3 End-to-End Performance

Figure 4b and Figure 5b present the end-to-end speedup distribution for the SPCG solver using ILU(0) and ILU(K) preconditioners, respectively. In the end-to-end analysis, we focus exclusively on converging matrices, as only these allow us to fully evaluate the effectiveness of both the factorization and solving phases, ensuring that the solution is reached within a predictable time. The residual accuracy for the convergence of a linear system is selected to be 1×10^{-12} and the maximum iterations allowed is 1000 iterations.

For SPCG-ILU(0), the end-to-end experiments demonstrate that sparsification provides a gmean speedup of 1.68×, ranging between 0.69–9.61× across converging matrices. This reflects the ability of the sparsification that does not affect or has a minimal effect on the convergence of the linear systems. Particularly, in 94.65% of the linear systems, the number of iterations stays approximately the same as the non-sparsified PCG. Given the higher per-iteration speedup of SPCG, an end-to-end speedup is achieved. A stronger trend is also observed in SPCG-ILU(K), where it provides a gmean end-to-end speedup of 3.73× over the non-sparsified PCG. The number of iterations stays approximately the same in 91.61% of matrices. Overall, it is shown the wavefront-aware sparsification algorithm demonstrates its effectiveness with both ILU(0) and ILU(K).

4.4 Ablation Study

This section discusses the effect of sparsification on the performance of preconditioning construction and solving phases as well as shows the efficiency of the wavefront-aware sparsification algorithm. In this section, we define the "Oracle" version as the fastest implementation achieved by selecting among three sparsified thresholds: 1%, 5%, and 10%. We assume this "Oracle" version represents the upper bound of sparsification performance for each matrix.

4.4.1 Sparsification effect on preconditioning and solve phases. To better understand how sparsification contributes to the overall process, we decouple the factorization phase and the PCG solving phase from each other and analyze each phase separately.

Figure 6 illustrates the effect of sparsification on the ILU(0) factorization phase for different sparsification levels: 1%, 5%, and 10%. The plot shows that the factorization is improved for most matrices across various sparsification levels, with higher levels slightly tending to achieve a greater speedup. This is because the sparsification alters the matrix structure by removing elements before factorization, which changes the paths through which dependences propagate. This reduces the number of wavefronts and requires fewer operations during factorization, optimizing both memory access patterns and arithmetic intensity.

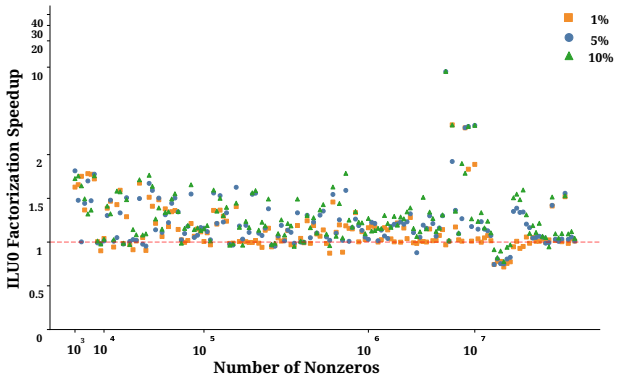


Figure 6: Sparsified ILU(0) factorization speedup on A100.

For the impact on the PCG solving phase, Table 1a presents the per-iteration speedup of SPCG-ILU(0) across different sparsification ratios for all matrices. The geometric mean speedup for SPCG-ILU(0) shows consistent improvements, ranging from 0.98× to 1.22× across the three sparsification ratios. Using wavefront-aware sparsification in SPCG, the speedup reaches to 1.23×.

Similarly, as shown in Table 1b, SPCG-ILU(K) demonstrates notable improvements, with geometric mean speedups ranging from 1.47× to 1.65× for each individual ratio, confirming that sparsification significantly enhances iteration efficiency. These results show

Table 1: Per-iteration speedup of SPCG over PCG on A100.

(a) Per-iteration speedup statistics of SPCG-ILU(0).

Statistic/Setting	1%	5%	10%	SPCG	Oracle
Geometric Mean	0.98×	1.11×	1.22×	1.23×	1.39×
% Accelerated	56.14%	71.93%	68.42%	69.16%	78.07%

(b) Per-iteration speedup statistics of SPCG-ILU(K).

Statistic/Setting	1%	5%	10%	SPCG	Oracle
Geometric Mean	1.47×	1.62×	1.65×	1.65×	1.78×
% Accelerated	88.57%	92.86%	85.71%	80.38%	97.14%

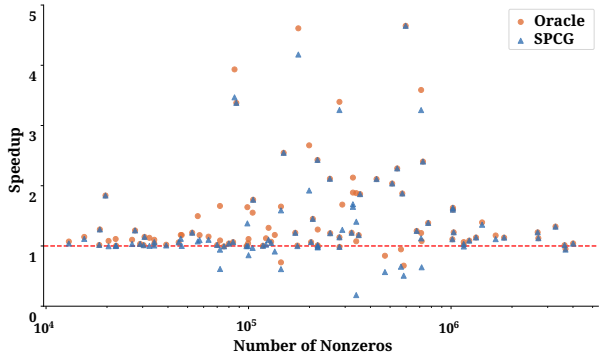


Figure 7: Per-iteration speedups of SPCG and the oracle implementations of SPCG-ILU(K)

that while higher sparsification levels generally improve computational efficiency for both preconditioners, this is not always the case. As observed in both tables, while applying a sparsification ratio of 10% yields a higher geometric mean speedup, a larger percentage of matrices achieve speedup with a sparsification ratio of 5%. This phenomenon can be attributed to the excessive wavefronts potentially introduced by over-aggressive sparsification, which negatively impacts performance. So even though for some matrices, 10% sparsification can achieve a higher speedup, a wider range of matrices benefit from a more moderate sparsification like 5%. This experiment highlights that sparsification effectively accelerates the iterative solver in both preconditioner and solve phases.

4.4.2 The effect of wavefront-aware sparsification. Additionally, as shown in Table 1, the oracle version achieves higher geometric mean speedups of 1.39× and 1.78×, highlighting the upper-bound potential of sparsification for accelerating the PCG process when the most suitable sparsification is applied.

Lastly, the overlapping of data points from SPCG and oracle variants illustrated in Figure 7 shows the effectiveness of the wavefront-aware sparsification algorithm. Specifically, for per-iteration performance and end-to-end performance respectively, 56.14% and 31.43% of the sparsified matrix selections match the oracle choices.

By comparing the data points from group of the SPCG variant in Figure 7 and formerly shown Figure 5b, we observe a slight upward shift of the data point distribution. This suggests that the end-to-end performance gains are not only from the PCG performance boost, but also from the speedups from the factorization phase as previously illustrated in Figure 6.

4.5 Portability

This section presents a cross-architecture consistency demonstration and analysis, highlighting the portability and performance of SPCG by applying SPCG-ILU(0) on a V100 GPU and EPYC CPU.

Table 2: Per-iteration speedup on A100 and V100 for SPCG-ILU(0) and SPCG-ILU(K).

Statistic/Setting	SPCG-ILU(0)		SPCG-ILU(K)	
	A100	V100	A100	V100
Geometric Mean	1.23×	1.22×	1.65×	1.71×
% Accelerated	69.16%	83.18%	80.38%	82.25%

To confirm the cross-architecture consistency by comparing the speedup effects of SPCG-ILU(0) on A100 and V100 GPUs, we conducted the same PCG tasks on both architectures across the same set of matrices. As shown in Table 2, the measured speedups for SPCG-ILU(0) are 1.23× on A100 and 1.22× on V100, while SPCG-ILU(K) yields 1.65× on A100 and 1.71× on V100, demonstrating that both GPUs consistently benefit from sparsification.

It is important to clarify that a higher speedup on one device compared to another does not necessarily indicate which GPU is faster overall, because speedup is a relative term between the sparsified version and the baseline, instead of an absolute timing metric. It reflects each device's ability to leverage improved parallelism in SPCG. One primary reason for the observed speedups is that reduced wavefronts enable higher parallel execution. Consequently, variations in speedup magnitude arise from device-specific factors, including architectural differences in memory bandwidth, parallel execution units, and thread-block organization, which can either amplify or mitigate the performance impact of reduced wavefronts.

Specifically, the number of wavefronts in SPCG-ILU(0) consistently decreases and memory constraints are minimal due to zero fill-ins. As a result, GPUs with a higher number of cores, such as the A100, achieve slightly better geometric mean speedups compared to the V100. This improvement is attributed to the A100's abundant resources, which enhances its ability to exploit the reduced wavefronts effectively. Conversely, in SPCG-ILU(K) scenarios, fill-ins introduce variability and memory can become a bottleneck. In these cases, the V100, despite having smaller shared memory resources than the A100, benefits more significantly from SPCG's reduction of the solver's memory footprint. This reduction alleviates memory bottlenecks more effectively on the V100, resulting in comparatively larger performance gains than those observed on the A100.

Examining the speedup distributions illustrated in Figure 8a-8b, the histograms reveal that most speedup values exceed 1. This indicates that, under the V100 scenario, the majority of matrices experience moderate and stable performance improvements. Additionally, any performance degradations observed in some matrices remain negligible. These findings are consistent with the speedup effects observed on the A100 architecture, as shown in Figure 4a and Figure 5a, further demonstrating cross-architecture consistency.

While the focus of the work is on GPUs, we tested the solve phase of SPCG-ILU(0) on an AMD EPYC CPU as shown in Figure 8c. SPCG improves the performance on CPU by a gmean per-iteration speedup of 1.24×, with 91.59% of matrices benefiting from the sparsification. This clearly shows the wavefront parallelism improvement helps CPU architectures as well as GPUs.

4.6 Low-rank Approximation Methods

Low-rank approximation methods apply sparsification to the factors of preconditioners or factorizations in linear solvers where local low-rank structures exist. This technique is fundamentally different from SPCG, which is applied to the input matrix regardless of rank structures and is instead based on the magnitude of values. Identifying low-rank structures, especially in incomplete solvers, is challenging because their frontal or supernodal matrices—dense sub-matrices that hold fill-ins in the factors—are small, thus offering fewer opportunities for low-rank approximation.

We use STRUMPACK [22] to explore low-rank solvers and their different parameters. Particularly, we focus on Hierarchically Semi-Separable (HSS) low-rank matrices and their various compression parameters, including compression leaf size, relative and absolute compression tolerances, and minimum separator size. However, low-rank approximations were rarely triggered for incomplete solvers. Despite experimenting with sweeping leaf size and compression tolerances, we observed that HSS compression was only effectively applied in 5.61% of the tested matrices. Reducing the minimum separator size noticeably increased HSS usage, raising the coverage to 28.04%. However, this setting negatively impacted both performance and memory usage and is therefore not recommended.

Therefore, our analysis suggests that the majority ILU(0) or ILU(K) factors are not qualified to trigger HSS compression from STRUMPACK. Even when direct solver, i.e., LU factorization is used as preconditioner, low-rank approximation does not improve the performance. Due to this methodological mismatch, comparing SPCG directly with STRUMPACK becomes less insightful.

5 ANALYSIS

This sections illustrates a series of analyses on application, matrix, and architecture characteristics to gain a better understanding of when and why sparsification is beneficial.

5.1 Impact of Application Characteristics

Based on experiment results, we observe that the application category is a useful indicator for deciding whether to apply sparsification, as matrices from the same categories often share properties determined by the nature of the problem.

Figure 9 presents bar charts showing the geometric mean end-to-end speedups across 17 application categories. Generally, 16 of these categories show either moderate or strong improvements on end-to-end performance, demonstrating a constant boost brought by SPCG. To understand where the end-to-end performance gains originate and where they deteriorate, we further analyze these results by considering per-iteration speedups.

Among the categories that exhibit good end-to-end speedups, several stand out for their consistent gains. In particular, *economic*, *duplicate optimization*, and *circuit simulation* show strong improvements in overall runtime. These categories also report excellent per-iteration speedups, suggesting that sparsification effectively reduces wavefront depth and enables better parallelism in triangular solves within each CG iteration. The preconditioners generated in these cases not only support faster iteration steps but also preserve convergence behavior, resulting in compounded and cumulated benefits across the full solve process.

On the other hand, categories such as *computational fluid dynamics* and *computer graphics/vision problems* show relatively moderate end-to-end improvements, despite achieving substantial per-iteration speedups. This mismatch suggests that although the cost of each iteration is reduced, the number of iterations required increases. In these cases, the original acceleration advantage is diluted by degraded convergence, leading to only partial gains overall.

5.2 Wavefront Analysis

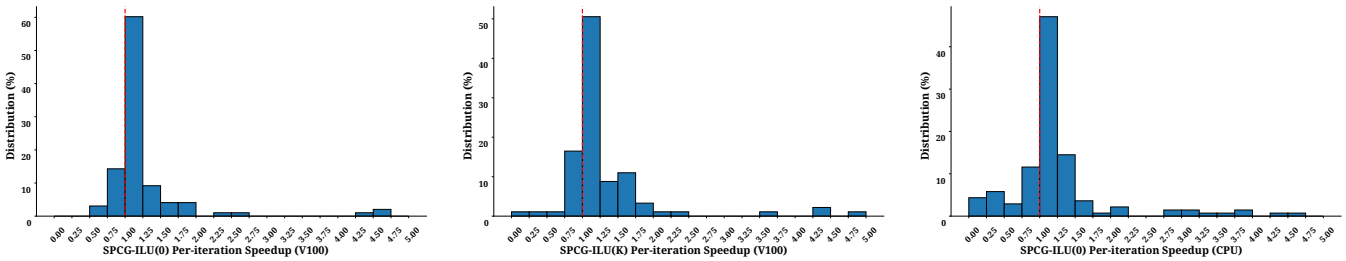
We now consider the effect of sparsification on number of wavefronts. Figure 10a displays the results of analyzing whether wavefront reduction and per-iteration speedup are correlated for SPCG-ILU(0). The wavefront reduction ratio defined in Equation 7 is utilized to evaluate the correlation. A trendline is computed using linear regression to illustrate how wavefront reduction impacts per-iteration speedup. It visually demonstrates a positive correlation between wavefront reduction and per-iteration speedup. A Spearman's correlation coefficient of 0.61 also reflects this moderately strong correlation. The correlation highlights that reducing the wavefront through sparsification leads to significant improvements in parallelism during the PCG solving phase, which is the primary reason sparsification enhances per-iteration solving performance.

The strength of this correlation arises from the fundamental impact of the number of wavefronts on parallel performance of preconditioners and triangular solvers. Reducing dependences results in fewer wavefronts, leading to faster computations. By reducing the number of wavefronts, parallelism is improved by reduced synchronization and increased parallelism within each wavefront. Although the exact degree of speedup varies depending on the matrix and sparsification level, the clear relationship between wavefront reduction and speedup is consistent across the dataset.

Similarly, Figure 10b shows a correlation between wavefront reduction and per-iteration speedup for ILU(K). As the wavefront reduction ratio increases, the per-iteration speedup tends to increase, confirming the link between these two metrics in this scenario, too. For SPCG-ILU(K), sparsification achieves more efficient memory access patterns and computational reductions also by reducing the number of wavefront elements processed during each iteration. However, due to the interplay between sparsification and fill-ins introduced by ILU(K), the effect of sparsification on wavefront reduction is more complex and thereby less strong than in ILU(0), achieving a Spearman's correlation coefficient of 0.22, indicating a positive but weaker correlation.

5.3 GPU Profiling Observations

To gain deeper insights into the performance implications of sparsification on GPU hardware, we employed NVIDIA Nsight Compute to profile matrices. We discuss three representative matrices:



(a) SPCG-ILU(0) on V100. (b) SPCG-ILU(K) on V100. (c) SPCG-ILU(0) on AMD EPYC CPU.

Figure 8: The distribution of per-iteration speedups by SPCG on V100 GPU and on AMD EPYC 7413 CPU

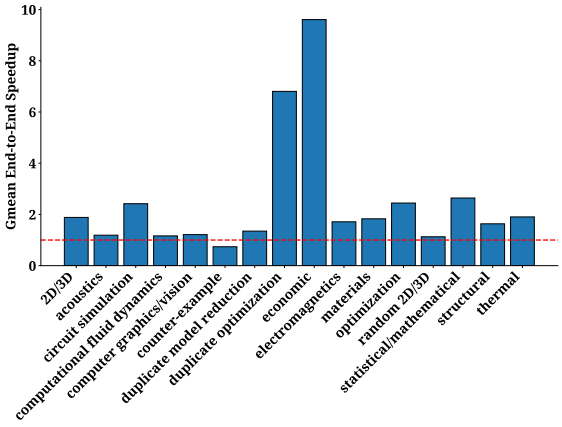
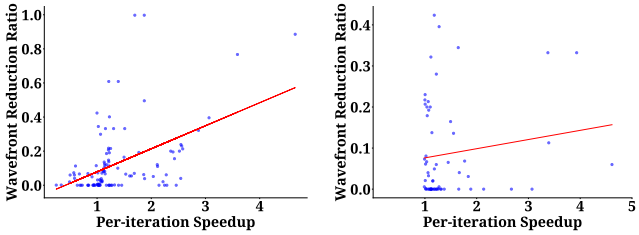


Figure 9: SPCG-ILU(0) speedup over PCG across applications.



(a) SPCG-ILU(0) (b) SPCG-ILU(K)

Figure 10: Correlation of wavefront reduction with per-iteration speedup for ILU preconditioners applied with SPCG.

thermomech_dM, *2cubes_sphere*, and *Muu*. These matrices, each serving as a representative example of a broader category, exhibit varying levels of speedup due to sparsification.

5.3.1 Memory-Bandwidth Utilization: Sparsification helps reduce global memory usage, resulting in improved DRAM throughput. This reduction in memory bandwidth bottlenecks contributes directly to the observed speedups, as the GPU cores spend less time waiting for memory operations to complete. For example, DRAM utilization decreased from 1.71% to 1.07% for *Muu*, contributing to its modest speedup of 0.99×. Conversely, *thermomech_dM* demonstrated an increase in DRAM utilization from 4.24% to 6.25%, correlating with its speedup of 4.39×.

5.3.2 Shift Towards Compute-Bound Behavior: For matrices where sparsification leads to performance improvements, we observed a transition from memory-bound to compute-bound behavior. This shift occurs because the reduced memory footprint allows the GPU to better utilize its arithmetic and logic units, thereby enhancing overall computational efficiency. A notable example is *thermomech_dM*, where compute utilization increased from 16.49% to 23.71%, reflecting better use of compute resources. In contrast, *2cubes_sphere* exhibited constant compute utilization of 1.07% before and after sparsification, indicating its performance is more reliant on memory latency than compute efficiency. These findings highlight the impact of sparsification: reducing global memory pressure and improving compute-resource utilization. However, the degree of benefit varies based on matrix characteristics, underscoring the importance of careful parameter tuning for sparsification.

5.4 Condition Number Analysis

The condition number of a matrix provides a key measure of its numerical stability and sensitivity, particularly in the context of iterative solvers like SPCG. For a linear system $Ax = b$, it quantifies how errors in the input vector b can amplify in the solution x. Specifically, the condition number is the maximum ratio of the relative error in x to the relative error in b. A high condition number implies that small inaccuracies in b may cause significant errors in x, while a low condition number indicates that the solution x will remain relatively stable. The condition number is an intrinsic property of the matrix, independent of the machine's floating-point precision or the algorithm used to solve the system [8]. This makes it a critical factor in understanding how sparsification influences convergence behavior in PCG solvers.

From the analysis in previous sections, sparsification has demonstrated the potential to improve PCG solvers by reducing time per-iteration through decreased wavefronts. Beyond this computational benefit, we found that 24 matrices in our dataset exhibited cases where sparsification enhanced convergence. To explore the underlying reasons, we examined the condition numbers of these matrices and their sparsified variations at ratios of 1%, 5%, and 10%. Representative examples include the matrices *ecology2*, *thermal1*, and *Pres_Poisson*, each illustrating a distinct pattern of behavior.

The system *ecology2* fails to converge in 1000 iterations without sparsification or at 1%, with final residuals above 1. At 5% and 10%, however, it converges in 2 iterations, reaching residuals as low as 1.2×10^{-13}. This dramatic improvement aligns with a condition number drop from 30 to 10, confirming the enhanced convergence.

In *thermal*1, the effect of sparsification is gradual but consistent. As the sparsification ratio increases, iteration counts drop steadily from over 1000 to 531, then 127, and finally 71 at 10%. The condition number also declines from 10.71 to 10.70 and then to 10.61, indicating improved conditioning and accelerated convergence.

Pres_Poisson shows diminishing returns at higher sparsification levels. Convergence improves up to 5%, with iterations dropping from 458 to 401 and the condition number from 1.11×10^4 to 1.07×10^4. However, at 10%, the system fails to converge within 1000 iterations, and the condition number returns to 1.11×10^4. This suggests that excessive sparsification can remove structurally critical entries, degrading conditioning and weakening the preconditioner.

In conclusion, appropriate sparsification can enhance convergence and computational efficiency, as confirmed by corresponding improvements in condition numbers. However, excessive sparsification may lead to a deterioration in both convergence and stability.

6 RELATED WORK

Iterative solvers such as PCG are known to be important components in several scientific simulations and are used as a way to measure the efficiency of systems for scientific applications. Dongarra et al. [19] proposed and developed the High-Performance Conjugate Gradient benchmark (HPCG) to assess high-performance computing systems. HPCG focuses on common data access patterns in various scientific applications. We discuss two major categories of accelerating PCG: wavefront parallelism and approximation.

6.1 Wavefront Techniques to Accelerate PCG

Using wavefront techniques to improve the performance of loop-carried sparse dependence, such as those in sparse triangular solvers and sparse incomplete LU factorization, is a common approach because these techniques do not affect accuracy. Several DAG scheduling techniques have been proposed to leverage wavefront parallelism on CPU architectures. Wavefront parallelism methods [2, 3, 12–15, 37, 44] first create a dependency DAG, and then compute a list of wavefronts to benefit from partial parallelism.

While wavefront parallelism ensures thread-level parallelism, warp-level parallelism is also essential for GPUs where abundant parallel resources exist. A wide range of scheduling techniques applied to sparse triangular solvers [34, 36, 41] or incomplete LU factorization [5, 11] to leverage both thread-level and warp-level parallelism on GPUs. cuSPARSE [36] is the state-of-the-art NVIDIA library. CapelliniSpTRSV [41] and synchronization-free SpTRSV [34] use atomic operations and busy-waiting to mitigate the impact of barrier synchronization in wavefront parallelism. ILU preconditioners also benefit from warp-level parallelism and address the dynamic nature of ILU using multiple fixed-point iterations [5] or block structures [11]. However, the dependences between iterations remain the main obstacle to fully utilizing parallel resources, wasting GPU time. Sparsification can reduce the number of wavefronts and synchronizations. SPCG leverages existing implementations and improves their performance on GPUs and CPUs.

6.2 Approximation in Linear Solvers

Matrix factorization methods, such as Gaussian elimination, offer a general and direct way to solve linear systems. Several direct solvers have been developed [17, 35] but their scalability is limited by memory and computation requirements of fill-ins in their factors. Therefore, approximation techniques are used to compute an estimates of the factors with reduced memory and computational requirements. These approximate factors are then used with preconditioned iterative solvers, such as PCG. We classify approximation methods for linear solvers into three categories: mixed-precision methods [1, 24], factorized sparse approximate inverse (FSAI) methods [4, 9], and sparsification methods [22, 28, 30].

Mixed-precision approaches [1, 21, 23–25, 27, 32, 42, 45] use low-precision data types, such as `float16`, to reduce the size of the factors. Mixed-precision linear solvers are especially important for GPUs, where low-precision data types are commonly supported and lead to less expensive computations. Mixed-precision direct linear solvers are often used in preconditioned iterative solvers [24]. The SPCG solver proposed in this work can additionally benefit from mixed-precision design.

The sparse approximate inverse (SAI) [9, 16] is another approximation method for solving linear systems through preconditioning. The SAI preconditioner is based on assumption that a sparse approximate inverse of a given sparse matrix exists. The assumption of sparsity in the approximate inverse is because the magnitude of values in the actual inverse is often small. Therefore, the inverse is approximately computed for a given sparsity pattern. The sparsity pattern can be determined statically [4] or dynamically [9]. Recent SAI techniques [29] also specify sparsity patterns that are computable with GPU architectures and their tensor cores. SAI techniques rely on matrix multiplication to apply the approximate inverse, enabling efficient use of GPU resources. While SAI techniques are efficient, not all matrices have a sparse approximate inverse, limiting the generality of these techniques.

Sparsification is a common approach for approximating linear system solvers and creating preconditioners to maintain generality and efficiency. Application-specific sparsification, particularly for Laplacian diagonally dominant matrices [28, 30, 38], has received considerable attention. These methods sparsify fill-in based on application properties. While efficient, these algorithms are specific to a particular class of applications. Rank information is commonly used to reduce computation. Many sparse matrices are full-rank but contain several zero blocks. During factorization, these blocks can be filled with nonzero elements exhibiting low-rank properties in some parts, allowing for the removal of some elements from the factor. Hierarchical low-rank approximation methods (H-Matrices) [7, 33, 43] can extract local low-rank properties in certain regions of the frontal matrices. H-matrices are used in sparse linear solvers by applying it to tiles [10] or frontal matrices [22, 39] during factorization. STRUMPACK [22] leverages this concept, using various H-matrix and compression algorithms to sparsify the factors. To be efficient, these methods require large frontal matrices, which occur in direct factorization.

The most commonly used sparsification method in solving linear systems is incomplete solvers, where the number of fill-ins is pre-determined to control memory usage. Some examples are ILU(K), and Incomplete Cholesky with K fill-in (IC(K)) solvers, which are commonly used as preconditioners in iterative solvers such as PCG [20]. This category is general and relies on reducing the number of fill-ins based on the value of K during factorization. Several efficient incomplete solvers [36] are implemented for GPUs. This sparsification methodology is the closest to our own. The problem with sparsification in these incomplete solvers is that they still retain many fill-ins that are not essential [39]. Even when the dropping threshold is changed, most fill-ins remain in the factors. While low-rank approximation in STRUMPACK [22] and other methods [39] are also applied to incomplete solvers, GPU versions of these incomplete solvers are not supported; only a direct solver is implemented on GPUs. Also, low-rank methods work efficiently when the number of fill-ins is high, which is often not the case for incomplete solvers with zero or a small number of fill-ins.

7 SUMMARY AND CONCLUSION

This paper has described SPCG, which uses wavefront-aware sparsification to optimize incomplete LU preconditioners ILU(0) and ILU(K) on SPD matrices. SPCG improves the performance of the PCG solver with minimal effect on convergence. As a result, two variants of SPCG-ILU(0) and SPCG-ILU(K) are introduced, where the difference comes from the type of preconditioner. SPCG improves the efficient use of parallel resources by exposing more parallelism and reducing the cost of data movement. SPCG is thoroughly benchmarked across a wide range of SPD matrices. The sparsification in SPCG-ILU(0) and SPCG-ILU(K) leads to a geometric mean (gmean) speedup of 1.23× and 1.65× on an A100 GPU, respectively. SPCG also improves the end-to-end performance of linear solvers by a gmean of 1.68× and 3.73× on an A100 GPU for SPCG-ILU(0) and SPCG-ILU(K), respectively. SPCG demonstrates a similar speedup trend on a V100 GPU and an AMD EPYC CPU. Additional analysis was also conducted, showing the effect of the application and wavefronts on SPCG's performance.

ACKNOWLEDGMENTS

This work is supported by NSERC discovery grants (RGPIN-2023-04897, DGECR-2023-00133), NSERC alliance grant (ALLRP 586319-23), Chameleon cloud [26], and the Digital Research Alliance of Canada (www.alliancecan.ca).

REFERENCES

[1] Khalid Ahmad. 2024. *Data-driven Techniques to Accelerate Sparse Computations on Graphical Processing Units (GPUs).* The University of Utah.
[2] Jayvant Anantpur and R Govindarajan. 2013. Runtime dependence computation and execution of loops on heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* IEEE, 1–10.
[3] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1, 01 (1989), 73–95.
[4] Hartwig Anzt, Edmond Chow, Thomas Huckle, and Jack Dongarra. 2016. Batched generation of incomplete sparse approximate inverses on GPUs. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA).* IEEE, 49–56.
[5] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. 2019. ParILUT-a parallel threshold ILU for GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 231–241.
[6] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1994. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM.
[7] Mario Bebendorf. 2008. *Hierarchical matrices.* Springer.
[8] David A Belsley, Edwin Kuh, and Roy E Welsch. 2005. *Regression diagnostics: Identifying influential data and sources of collinearity.* John Wiley & Sons.
[9] Massimo Bernaschi, Mauro Carrozzo, Andrea Franceschini, and Carlo Janna. 2019. A dynamic pattern factored sparse approximate inverse preconditioner on graphics processing units. *SIAM Journal on Scientific Computing* 41, 3 (2019), C139–C160.
[10] Wajih Boukaram, Stefano Zampini, George Turkiyyah, and David E Keyes. 2024. Cholesky Factorization of Tile Low Rank Matrices on GPUs. (2024).
[11] Yan Chen, Xuhong Tian, Hui Liu, Zhangxin Chen, Bo Yang, Wenyuan Liao, Peng Zhang, Ruijian He, and Min Yang. 2018. Parallel ILU preconditioners in GPU computation. *Soft Computing* 22 (2018), 8187–8205.
[12] Kazem Cheshmi. 2022. *Transforming Sparse Matrix Computations.* Ph. D. Dissertation. University of Toronto (Canada).
[13] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–13.
[14] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2018. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 779–793.
[15] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23).* Association for Computing Machinery, New York, NY, USA, Article 89, 15 pages. https://doi.org/10.1145/3581784.3607097
[16] Edmond Chow. 2001. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *The International Journal of High Performance Computing Applications* 15, 1 (2001), 56–74.
[17] Timothy A Davis. 2008. User Guide for CHOLMOD: a sparse Cholesky factorization and modification package. *Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA* (2008).
[18] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.
[19] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10.
[20] Sergiy Yu Fialko and Filip Zeglen. 2016. Preconditioned conjugate gradient method for solution of large finite element problems on CPU and GPU. *Journal of Telecommunications and Information Technology* 2 (2016), 26–33.
[21] Goran Flegar, Hartwig Anzt, Terry Cojean, and Enrique S Quintana-Orti. 2021. Adaptive precision block-Jacobi for high performance preconditioning in the Ginkgo linear algebra software. *ACM Transactions on Mathematical Software (TOMS)* 47, 2 (2021), 1–28.
[22] Pieter Ghysels and Ryan Synk. 2022. High performance sparse multifrontal solvers on modern GPUs. *Parallel Comput.* 110 (2022), 102897.
[23] Fritz Göbel, Thomas Grützmacher, Tobias Ribizel, and Hartwig Anzt. 2021. Mixed precision incomplete and factorized sparse approximate inverse preconditioning on GPUs. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27.* Springer, 550–564.
[24] Nicholas J Higham and Theo Mary. 2022. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31 (2022), 347–414.
[25] Takuya Ina, Yasuhiro Idomura, Toshiyuki Imamura, Susumu Yamashita, and Naoyuki Onodera. 2021. Iterative methods with mixed-precision preconditioning for ill-conditioned linear systems in multiphase CFD simulations. In *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA).* IEEE, 1–8.
[26] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20).* USENIX Association.
[27] Daniel Kressner, Yuxin Ma, and Meiyue Shao. 2023. A mixed precision LOBPCG algorithm. *Numerical Algorithms* 94, 4 (2023), 1653–1671.
[28] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A Spielman. 2016. Sparsified cholesky and multigrid solvers for connection laplacians. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing.* 842–850.

[29] Sergi Laut, Ricard Borrell, and Marc Casas. 2024. Extending Sparse Patterns to Improve Inverse Preconditioning on GPU Architectures. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 200–213.

[30] Yin Tat Lee, Richard Peng, and Daniel A Spielman. 2015. Sparsified cholesky solvers for SDD linear systems. *arXiv preprint arXiv:1506.08204* (2015).

[31] Xiaoye S Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 302–325.

[32] Neil Lindquist, Piotr Luszczek, and Jack Dongarra. 2021. Accelerating restarted GMRES with mixed precision arithmetic. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 1027–1037.

[33] Bangtian Liu, Kazem Cheshmi, Saeed Soori, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2020. MatRox: modular approach for improving data locality in hierarchical (Mat)rix App(Rox)imation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 389–402. https://doi.org/10.1145/3332466.3374548

[34] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. 2016. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 617–630.

[35] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[36] Maxim Naumov. 2011. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011* 1 (2011).

[37] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*. Springer, 124–140.

[38] Richard Peng and Daniel A Spielman. 2014. An efficient parallel solver for SDD linear systems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. 333–342.

[39] Hadi Pouransari, Pieter Coulier, and Eric Darve. 2017. Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. *SIAM Journal on Scientific Computing* 39, 3 (2017), A797–A830.

[40] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[41] Jiya Su, Feng Zhang, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2020. CapelliniSpTRSV: a thread-level synchronization-free sparse triangular solve on GPUs. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[42] Zhen Xiao, Tong-Xiang Gu, Yuan-Xi Peng, Xiao-Guang Ren, and Jin Qi. 2018. Mixed precision in CUDA polynomial precondition for iterative solver. In *2018 IEEE International Conference on Computer and Communication Engineering Technology (CCET)*. IEEE, 186–192.

[43] Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. 2017. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[44] Behrooz Zarebavani, Kazem Cheshmi, Bangtian Liu, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2022. HDagg: Hybrid Aggregation of Loop-carried Dependence Iterations in Sparse Matrix Computations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1217–1227. https://doi.org/10.1109/IPDPS53621.2022.00121

[45] Haoyuan Zhang, Wenpeng Ma, Wu Yuan, Jian Zhang, and Zhonghua Lu. 2024. Mixed-precision block incomplete sparse approximate preconditioner on Tensor core. *CCF Transactions on High Performance Computing* 6, 1 (2024), 54–67.