

Argus: An Efficient Accelerator Design for Sparse Nonlinear Solver

Fangzhou Ye

University of Central Florida
Orlando, FL, USA
fangzhou.ye@ucf.edu

Amir Ghazizadeh Ahsaei

University of Central Florida
Orlando, FL, USA
amir.g@ucf.edu

Hao Zheng

University of Central Florida
Orlando, FL, USA
hao.zheng@ucf.edu

Abstract—Nonlinear equations are a fundamental computational primitive in many scientific and engineering applications. The Newton-Raphson method is widely used for solving nonlinear equations due to its strong convergence behavior and broad applicability. However, nonlinear solvers require repeated evaluation of the first-order partial derivatives of the nonlinear equations (i.e., the Jacobian matrix), leading to dynamic and irregular sparsity patterns across iterations. The evolving sparsity patterns require the Jacobian to be reloaded and recomputed at each iteration, reducing the effectiveness of existing accelerator designs for linear solvers and sparse-dense matrix multiplication (SpMM).

In this paper, we posit that decoupling the static components from the dynamic Jacobian matrix, rather than recomputing the entire matrix at every iteration, can effectively exploit both data reuse and parallelism. Based on this insight, we propose Argus, a specialized accelerator architecture built on a Schur complement-based reformulation of the Newton-Raphson method. Specifically, we reformulate Jacobian computation through the Schur complement to decouple static and dynamic sparsity patterns from the Newton-Raphson method, reducing the reload and recomputation of the Jacobian matrix over iterations. In addition, the reformulation also converts sparse matrix-vector multiplication (SpMV) of nonlinear computations to general matrix-vector multiplication (GEMV). We further propose a modified weighted k -core decomposition method and construct a new Coupled Dependency Graph to avoid communication bottlenecks while improving the data locality for Schur complement computations. Lastly, we design a novel accelerator architecture and dataflow mapping that can support both Schur complement and GEMV computations. Our evaluation shows that Argus delivers performance speedups of $1.86\times$, $5.32\times$, and $5.68\times$ over state-of-the-art iterative solver designs Azul [1], LevelST [2], and Telos [3], respectively.

Index Terms—Nonlinear solver, Newton-Raphson method, Schur complement, graph abstraction, accelerator.

I. INTRODUCTION

Nonlinear equations are a fundamental computational kernel in many scientific and engineering applications [4]–[6], including structural deformation analysis in civil engineering, simulation of complex chemical reactions, and computational fluid dynamics. Consequently, efficiently solving nonlinear equations is critical to advancing scientific discovery. Among the many approaches that have been developed, including both direct and iterative methods, the Newton-Raphson method [7]–[9] remains the most commonly used selection for large-scale, highly sparse nonlinear solvers because of its broad applicability and robust convergence behavior. Despite its

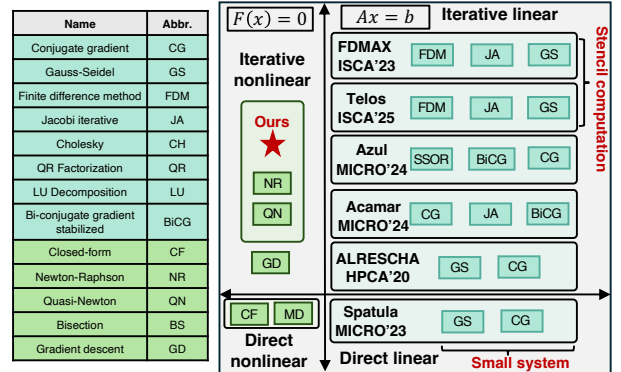


Fig. 1. Taxonomy of numerical solvers and prior state-of-the-art accelerator designs.

effectiveness, nonlinear iterative solving presents several computational challenges. In particular, the repeated evaluation of first-order partial derivatives requires frequent reloading and recomputation of the Jacobian matrix across iterations, resulting in poor data locality and limited parallelism.

Numerous prior designs [1]–[3], [10]–[13] accelerate iterative solvers by retaining frequently accessed data on chip to improve locality and throughput. However, these approaches typically assume either a fixed coefficient matrix or structured stencil patterns, and therefore have limited applicability to handle the evolving coefficient matrix (e.g., Jacobian matrix) in nonlinear solvers. For example, as summarized in Figure 1, Telos [3] and FDMAX [10] exploit structured stencil sparsity patterns inherent to finite difference discretizations when accelerating partial differential equations (PDEs). However, nonlinear solvers exhibit irregular and unstructured sparsity that evolves over iterations.

More general sparse accelerator designs have also been explored. Azul [1], for example, adopts a distributed-SRAM architecture combined with hypergraph-based scheduling to optimize on-chip data locality. Acamar [13] employs a dynamically reconfigurable architecture to adapt to diverse sparse inputs, optimizing the trade-off between resource utilization and latency. In parallel, recent efforts [2], [11] have targeted an alternative bottleneck in iterative solvers by accelerating sparse triangular solve (SpTRSV) during runtime. Alrescha [11] improves SpTRSV execution by separating parallelizable and inherently sequential computation regions. LevelST [2] adopts a stream-based execution model that reorganizes sparse matrices

TABLE I
NONLINEAR FUNCTIONS SOLVED BY NEWTON-RAPHSON AND THEIR CHARACTERISTICS

Application	Formula $F(x) = 0$	Type	Matrix Size
Nonlinear Poisson Equation	$-\nabla^2 u + u^3 = f$	Elliptic PDE	$10^4\text{--}10^6$
Incompressible Navier-Stokes	$(\mathbf{u} \cdot \nabla)\mathbf{u} - \nu \nabla^2 \mathbf{u} + \nabla p = \mathbf{f}, \nabla \cdot \mathbf{u} = 0$	Parabolic PDE	$10^5\text{--}10^7$
Nonlinear Elasticity	$\nabla \cdot \sigma(\mathbf{u}) + \mathbf{f} = 0, \sigma = \sigma(\nabla \mathbf{u})$	Elliptic PDE	$10^4\text{--}10^6$
Circuit Simulation	$\mathbf{f}(\mathbf{v}) = \mathbf{Y}\mathbf{v} + \mathbf{i}(\mathbf{v}) - \mathbf{s} = 0$	Algebraic system	$10^3\text{--}10^5$
Radiative Transfer Equation	$\Omega \cdot \nabla I + \kappa I = \kappa B(T), F(T) = \nabla \cdot \mathbf{q} + Q = 0$	Integro-differential PDE	$10^5\text{--}10^7$
Chemical Reaction Networks	$\frac{dc}{dt} = \mathbf{S}\mathbf{r}(c) = 0$	Nonlinear ODE	$10^2\text{--}10^4$
Magnetohydrodynamics	$\rho(\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + (\nabla \times \mathbf{B}) \times \mathbf{B}$	Coupled PDE system	$10^5\text{--}10^8$
Nonlinear Heat Conduction	$\nabla \cdot (k(T)\nabla T) + Q = 0$	Parabolic PDE	$10^4\text{--}10^6$

into sequential data tiles. However, the continually evolving Jacobian matrix requires repeated recomputation and data movement, making prior designs less effective for accelerating nonlinear solvers.

In this paper, we identify that the fundamental challenge in nonlinear solvers stems from the Jacobian matrix, whose numerical values and sparsity structure change dynamically across iterations. However, we observe that a substantial portion of the Jacobian matrix remains stable from one iteration to the next, exposing significant reuse opportunities that existing designs fail to exploit. In particular, when the residual can be decomposed into linear and nonlinear components, the linear operator is independent of the current iterate and therefore remains constant across Newton iterations. Leveraging this insight, we introduce a Schur complement-based reformulation of the Newton iteration that decomposes the large, dynamic Jacobian into smaller submatrices. This reformulation both (1) reduces Jacobian reloading and recomputation across iterations and (2) transforms the resulting computation from SpMV kernels into GEMV-based operations. Based on this reformulation, we propose Argus, a specialized accelerator architecture for sparse nonlinear solvers. Our main contributions are as follows:

- We identify that nonlinear solvers could be decomposed into linear and nonlinear components, where linear subsets can be optimized independently of the Newton iterations. This provides a new opportunity to avoid reload and recomputation of the Jacobian matrix over iterations, improving the parallelism and data locality. To the best of our knowledge, this is the first work to systematically characterize and exploit this reuse opportunity in the context of nonlinear solvers.
- We propose a Schur complement-based reformulation of the Newton-Raphson method that exposes the above reuse opportunities of nonlinear solvers. The proposed Schur complement-based reformulation reduces the size of the recalculating Jacobian matrix over iterations and converts sparse nonlinear computations to GEMV operations. In addition, we propose a weighted k -core graph decomposition algorithm that partitions the Jacobian into stable and unstable submatrices, where stable submatrices can be reused over iterations.
- We propose a dedicated accelerator architecture, Argus, that co-designs a hardware Jacobian-graph Scheduler, a

Algorithm 1 Newton Iterations with Preconditioned Conjugate Gradient (PCG)

Input: Initial guess x_0 , tolerance ε , max inner iterations m

Output: Approximate solution x^* to $f(x) = 0$

```

1:  $k \leftarrow 0$ 
2: while  $\|f(x_k)\| > \varepsilon$  do
3:    $J_k \leftarrow \nabla f(x_k)$    Outer iter. with dynamic Jacobian
4:    $b \leftarrow -f(x_k), \delta x \leftarrow 0, r_0 \leftarrow b - J_k \delta x$ 
5:   Solve  $Mz_0 = r_0$     $\triangleright$  Apply preconditioner
6:    $p_0 \leftarrow z_0$ 
7:   for  $j = 0, 1, \dots$  until convergence or  $j = m$  do
8:      $\alpha_j \leftarrow (r_j, z_j) / (J_k p_j, p_j)$     $\triangleright$  SpMV:  $J_k p_j$ 
9:      $\delta x \leftarrow \delta x + \alpha_j p_j$ 
10:     $r_{j+1} \leftarrow r_j - \alpha_j J_k p_j$ 
11:    if  $\|r_{j+1}\| \leq \varepsilon$  then
12:      break
13:    Solve  $Mz_{j+1} = r_{j+1}$    Inner iter. with linear equation
14:     $\beta_j \leftarrow (r_{j+1}, z_{j+1}) / (r_j, z_j)$ 
15:     $p_{j+1} \leftarrow z_{j+1} + \beta_j p_j$ 
16:     $x_{k+1} \leftarrow x_k + \delta x, k \leftarrow k + 1$ 
17: return  $x_k$ 

```

Schur Core for Schur complement computations, and a unified PE array. In addition, we propose a Coupled Dependency Graph to jointly exploit inter- and intra-iteration reuse across Newton iterations.

We evaluate the proposed Argus using various SuiteSparse datasets, and the evaluation results show that the Argus design can achieve performance speedups of $1.86\times$, $5.32\times$, and $5.68\times$ over state-of-the-art iterative solver designs Azul [1], LevelST [2], and Telos [3], respectively.

II. BACKGROUND

A. Nonlinear Equations

Many real-world scientific and engineering problems require solving nonlinear equations, i.e., finding a vector x such that $F(x) = \mathbf{0}$, where $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a vector-valued function whose components are nonlinear in the unknowns. As summarized in Table I, nonlinear systems are pervasive across a broad spectrum of application domains [14], [15]. For example, nonlinear functions are common in structural mechanics (e.g., simulating large deformations of a bridge under load), computational fluid dynamics (e.g., modeling turbulent airflow over a wing), circuit simulation (e.g., capturing transistor behavior at varying voltages), and many other domains. Unlike linear equations (e.g., $Ax = \mathbf{b}$, where A is a fixed coefficient matrix and $\mathbf{b}$ represents a constant vector), nonlinear equations arise when the governing equations depend on the unknown variables in a way that is not linear, such as through products of variables, higher-order terms, or nonlinear functions.

B. Newton-type Method

Newton-Raphson methods. For nonlinear equations, direct solution is typically infeasible because closed-form solutions [16] rarely exist and direct methods often impose prohibitive computational and memory overheads. As a result, iterative methods are the predominant practical approach.

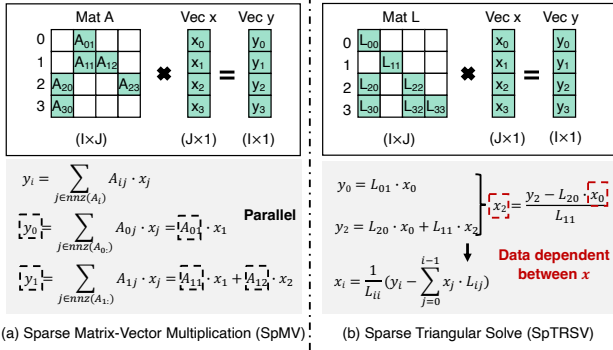


Fig. 2. Sparse matrix-vector multiplication and sparse triangular solve.

Starting from an initial value x_k , iterative methods progressively improve the solution until the residual norm $\|F(x_k)\|$ satisfies a prescribed convergence criterion. Rather than computing the solution in a single step, iterative methods generate a sequence of updates that gradually drive the residual toward zero. Among them, Newton-Raphson methods remain one of the most widely used methods due to their simplicity and effectiveness [7], [9]. At iteration k , Newton’s method linearizes F around the current iterate:

$$J(x_k) \delta x_k = -F(x_k), \quad x_{k+1} = x_k + \delta x_k, \quad (1)$$

where $J(x_k)$ denotes the Jacobian matrix, i.e., the matrix of first-order partial derivatives of F with respect to x , evaluated at the current iterate x_k . The Newton update δx_k is obtained by solving the resulting linear system. In many applications, the input matrices are sparse and are solved using iterative linear solvers such as Conjugate Gradient (CG) [17], BiCGSTAB [18], [19], and GMRES [20]. As shown in Algorithm 1, we use the preconditioned Conjugate Gradient (PCG) method as a representative example to illustrate how iterative linear solvers are employed within Newton’s method. Specifically, at each outer Newton iteration k , the Jacobian J_k is assembled and passed to PCG (Lines 7-15), which iteratively refines δx_k through a sequence of sparse matrix-vector multiplication (SpMV) operations on J_k until the residual falls below a prescribed tolerance (Lines 11-12). In this paper, we focus on Newton’s method with PCG, motivated by the widespread use of PCG-style sparse iterative solvers as a representative benchmark in high-performance computing [14], [21].

Other variants of Newton’s method, including *inexact Newton* methods [8], which solve the linear subproblem approximately, and *quasi-Newton* methods such as BFGS [22], which approximate J to avoid costly recomputation, trade solution accuracy for lower per-iteration overhead. Similarly, those variants of Newton’s methods share the same computational characteristics.

C. SpMV and SpTRSV

Sparse matrix-vector multiplication (SpMV) [23]–[29] computes the product of a sparse matrix A and a vector x , producing an output vector y . The computation of each element of y is illustrated in Figure 2(a). Since each product $A_{ij}x_j$

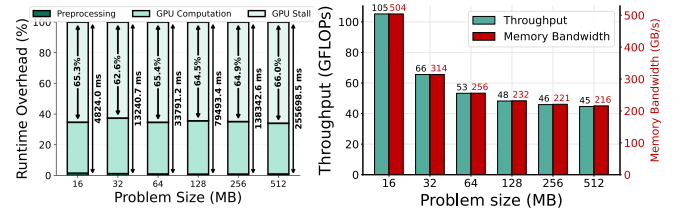


Fig. 3. Preliminary results of (a) End-to-end runtime breakdown and (b) Throughput performance and memory bandwidth by kernel of Newton-PCG on an A100 GPU across diverse problem sizes.

is independent, these operations can be performed in parallel. However, unlike a standalone SpMV operation, iterative solvers repeatedly reload the matrix A (i.e., the Jacobian matrix), introducing additional memory overhead, whereas SpMV reads each matrix element only once. Additionally, iterative solvers often require forward or backward substitution for triangular matrices, such as lower or upper triangular systems. Sparse triangular solve (SpTRSV) [1]–[3] is a fundamental linear algebra kernel for solving such triangular systems. As illustrated in Figure 2(b), for a lower triangular matrix L , the solver computes the first variable x_0 from the first equation, then proceeds sequentially by substituting known values to solve subsequent variables. This procedure creates strict data dependencies. For example, x_2 cannot be computed before x_0 , which severely limits parallelism.

D. Schur Complement

Schur complement [30] is a fundamental algebraic construction for the block partitioning of matrices. Given a block matrix of the form $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, the Schur complement S of the block A with respect to M is defined as $S = D - CA^{-1}B$, provided A is invertible. This construction applies block Gaussian elimination to reduce the problem sizes, effectively condensing the degrees of freedom associated with one block onto the remaining variables. Moreover, it translates sparse issues into sub-dense problems. Schur complements play a central role across numerical algebra and computational mathematics. For example, in the solution of large sparse linear systems, block elimination strategies exploit the Schur complement to reduce a global problem to a smaller, denser interface problem [31]. Beyond the application in linear systems, Schur complements are widely used in statistics (partial covariance), control theory (Riccati equations), and graph theory (effective resistance).

III. MOTIVATION AND CHALLENGES

A. Sparse and Large-scale Non-linear Systems

Nonlinear systems in large-scale scientific simulations often exhibit extreme sparsity and massive problem sizes, posing significant challenges for both computation and memory efficiency. This sparsity arises from the nature of many physical systems, where each unknown depends primarily on only a small set of neighboring variables. For example, as shown in Table I, in incompressible fluid-flow simulations over a fine mesh with millions of grid points [32], the resulting

Jacobian matrix can reach dimensions on the order of 10^7 , while each row contains only a small number of nonzero entries corresponding to neighboring grid points. To quantify resource utilization, we profile a Newton-PCG solver on an NVIDIA A100 PCIe GPU across six problem sizes ranging from 16 MB to 512 MB. As shown in Figure 3(b), both throughput and memory bandwidth decrease as the problem size increases. This degradation is caused by the repeated loading of the Jacobian matrix across iterations, indicating that simply scaling compute resources or memory bandwidth provides limited benefit for nonlinear solvers.

In some cases, the challenge is further exacerbated by changes in the sparsity pattern of the Jacobian matrix across Newton iterations, which break the opportunity to amortize scheduling or preprocessing costs over multiple iterations. For instance, consider a chemical reaction network modeling enzyme kinetics [33], [34], where certain enzymes become inactive when substrate concentrations fall below a threshold during the Newton iterations. Such deactivation removes the corresponding reaction pathways, causing Jacobian entries that were previously nonzero to vanish and to potentially reappear in later iterations as the iterate evolves.

Despite these challenges, we observe that a substantial portion of the Jacobian remains static across Newton iterations. In many large-scale nonlinear systems, variables are coupled through a mixture of linear and nonlinear terms, where only the nonlinear terms induce Jacobian entries that vary with the current iterate x_k . For example, in incompressible Navier-Stokes simulations over a fixed mesh [35], the momentum equation contains both a nonlinear convective term $(u \cdot \nabla)u$ and a linear viscous diffusion term $\nu \nabla^2 u$. The Jacobian entries associated with the viscous diffusion term are determined entirely by the mesh geometry and the viscosity coefficient ν , and therefore remain constant throughout Newton iterations regardless of how the velocity field evolves. As a result, a large fraction of the Jacobian’s nonzero structure is stable and can be reused across iterations, while only a relatively small dynamically changing region must be updated at each Newton step.

B. Limitations of Prior Works on Linear Solvers

Nonlinear iterative solvers exhibit computational characteristics that are fundamentally different from those of linear solvers, making them poorly suited to existing accelerator architectures designed for linear solvers, SpMM, or SpTRSV [1], [2], [11], [12], [36]–[44]. As shown in Algorithm 1, nonlinear solvers follow a nested loop structure, in which an *outer* Newton loop encloses an *inner* linear solver loop. At each outer Newton iteration, the Jacobian matrix must be computed before the next iteration can proceed. Repeated updates to the Jacobian matrix result in poor data locality and limited parallelism. As shown in Figure 3(a), GPU stall time becomes a major bottleneck in the end-to-end execution of the nonlinear solver. Note that GPU stall time includes both Jacobian computation time and GPU-CPU communication overhead, during which the GPU waits for the updated Jacobian matrix.

Figure 3(b) further shows that the nonlinear solver achieves at most 0.57% of peak throughput, even for the smallest problem sizes. Although prior work has extensively explored improving data locality and parallelism for sparse computations, these techniques have limited applicability to the new computational challenges introduced by dynamically evolving Jacobian matrices.

Specifically, Azul [1] retains the coefficient matrix A on chip across thousands of iterations to reduce DRAM traffic, and proposes a hypergraph-based data mapping strategy to reduce frequent communication. Spatula [45] targets sparse matrix factorization for direct linear solvers, proposing a hardware accelerator that exploits the elimination tree structure to improve scheduling efficiency and reduce fill-in overhead. Acamar [13] adopts a dynamically reconfigurable architecture that adjusts the SpMV unroll factor at runtime to adapt to diverse sparse input patterns, aiming to optimize the trade-off between resource utilization and latency. However, none of them is optimized for efficiently handling the dynamic evolution of the Jacobian matrix. On the other hand, many prior works discretize PDEs into linear systems using the finite difference method (FDM) [46], which approximates derivatives through differences between neighboring function values. FDMAX [10] supports reconfigurable grid sizes for solving PDEs using Jacobi or hybrid iterative solvers. Spadix [12] further presents a customized PE-array architecture for 3D PDE solving, exploiting spatial and temporal data reuse together with a checkerboard iterative method to improve convergence and reduce memory traffic. Telos [3] introduces a plane-parallel pipelining technique and hardware specialization, i.e., an ASIC design, to exploit structured sparsity patterns in PDE solving. Nevertheless, the optimization techniques of these approaches are only applicable to stencil-based computation patterns, which often lead to poor resource utilization and limited parallelism when applied to non-linear solvers.

In addition, because sparse triangular solve (SpTRSV) is a key bottleneck in iterative solvers [1], [3], other designs adopt vector architectures to accelerate SpTRSV and related iterative methods. These approaches typically assume fixed, statically analyzable computation patterns and rely on software scheduling to map independent computations onto vector units. For example, Alrescha [11] is a recent SpTRSV accelerator that separates parallelizable and non-parallelizable computation regions. LevelST [2] proposes a stream-based FPGA accelerator for SpTRSV that reorganizes the sparse matrix into sequential data tiles and resolves element dependencies in a streaming fashion, thereby avoiding irregular memory accesses and underutilized compute units.

Moreover, traditional SpMM accelerators [36]–[39] typically exploit data reuse through tiling and dataflow optimization. However, these techniques assume that both input matrices are known as a priority, which limits their effectiveness for iterative solvers, where the input vectors are updated at every iteration and computation proceeds in a streaming manner.

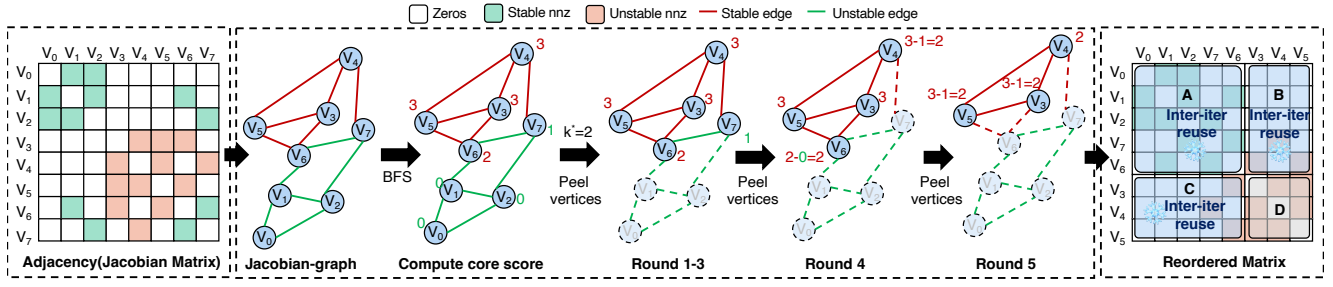


Fig. 4. Walk-through example of the proposed weighted k -core decomposition for Schur complement partition.

IV. SCHUR COMPLEMENT FOR NONLINEAR SYSTEMS

A. Schur Complement for Newton Iteration

While numerous accelerator designs have been proposed for sparse linear solvers [1]–[3], [10], [13] and SpMM acceleration [36], [37], these prior works are generally designed under the assumption that the coefficient matrix A remains fixed throughout the entire computation, allowing preprocessing overheads and on-chip storage costs to be amortized across thousands of iterations. This assumption, however, no longer holds for nonlinear solvers. As shown in Algorithm 1, Line 3, Newton-Raphson methods require recomputing the Jacobian $J(x_k)$ at every outer iteration k as the iterate x_k evolves.

Newton-type methods typically require tens to hundreds of outer iterations to converge [7], [9], each of which invokes an inner iterative linear solver that performs hundreds to thousands of SpMV operations [18], [23]. Consequently, the total number of Jacobian accesses from DRAM scales as $\mathcal{O}(K_{\text{outer}} \times K_{\text{inner}})$, making DRAM bandwidth the dominant performance bottleneck.

Key Insights. A substantial portion of the Jacobian can remain unchanged across Newton iterations when the nonlinear system contains both linear and nonlinear components. Specifically, the Jacobian contributions induced by linear terms are independent of the current Newton iterate and therefore remain constant, whereas only the contributions associated with nonlinear terms vary across iterations. For example, consider

$$F(x) = \begin{pmatrix} c_1 x_1 + c_2 x_2^2 \\ c_3 x_1^2 + c_4 x_2 \end{pmatrix},$$

Its Jacobian is

$$J(x) = \nabla F(x) = \begin{pmatrix} c_1 & 2c_2 x_2 \\ 2c_3 x_1 & c_4 \end{pmatrix}$$

In this case, the entries c_1 and c_4 are iteration-invariant because they derive from linear terms, while the off-diagonal entries vary with the current solution x . This example illustrates that, although the Jacobian is updated at every Newton iteration, part of its numerical content may remain constant and can therefore be reused across iterations.

Building on this insight, we develop a Schur complement-based decomposition of the Jacobian. After reordering the unknowns into two groups, $x = [x_A; x_D]$, the Newton-Raphson system in Equation 1 can be expressed in block form as:

$$J(x_k) \delta x_k = -F(x_k) \rightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \delta x_A \\ \delta x_D \end{bmatrix} = \begin{bmatrix} f_A \\ f_D \end{bmatrix} \quad (2)$$

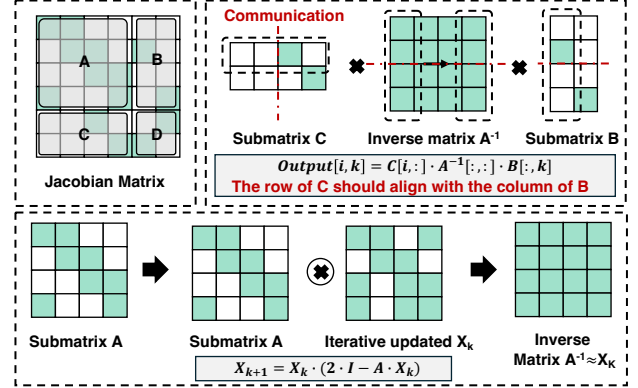


Fig. 5. The data dependencies of submatrices A , B , and C .

where A , B , C , D are blocks of Jacobian matrix, and $f = [f_A; f_D]$ (i.e., $f_A = -F_A(X^k)$, $f_D = -F_D(X^k)$). We aim to reorder unstable variables to block D while clustering stable constants to block A . Furthermore, given the use of the Schur complement, the computations associated with δx_A can be eliminated from Jacobian reconstruction. Consequently, the Jacobian computation can be simplified as:

$$\underbrace{(D - \underbrace{CA^{-1}B}_{\text{inter-iteration dataflow}})}_{\text{intra-iteration dataflow}} \delta x_D = f_D - CA^{-1}f_A, \quad (3)$$

where $S = D - CA^{-1}B$ is the *Schur complement* of the Jacobian [30]. Once δx_D is recovered from Equation 3, back-substitution yields $\delta x_A = A^{-1}(f_A - B \delta x_D)$. Please note that $CA^{-1}B$ is dense, and so is S . As such, we convert SpMV of nonlinear functions into GEMV operations. Furthermore, the direct calculation of the inverse matrix A^{-1} leads to an increased number of nonzeros (nnz), which are known as fill-in [45]. Therefore, we use an iterative method to calculate A^{-1} , as depicted in Figure 5, which requires fewer computations compared to direct methods. Because of the use of the Schur complement, we can (1) reduce the computational complexity of obtaining a new Jacobian matrix over Newton iterations and (2) simplify nonlinear iterative solvers to GEMV operations.

Accuracy Assurance. While holding fixed sub-matrices A , B , and C across Newton iterations enables inter-iteration reuse and substantially reduces DRAM traffic, it would cause the Schur complement S to drift from the true Jacobian, leading to degraded convergence rates and potentially unacceptable solution accuracy. To balance computational efficiency and numerical accuracy, we update A , B , and C every T Newton

Algorithm 2 Weighted k -Core Decomposition for Schur Complement Partition (WKCP)

Input: Jacobian $J \in \mathbb{R}^{n \times n}$, core threshold k^*

Output: Partition (V_A, V_D) of vertex set $V = \{1, \dots, n\}$

Step 1: Build Jacobian graph

1: $G \leftarrow \text{Adjacency}(J)$

Step 2: Assign edge stability weights

2: **for** Edge $e_{ij} \in E(G)$ **do**
 3: $w_{ij} \leftarrow 1$: stable edge, $w_{ij} \leftarrow 0$: unstable edge

Step 3: Compute weighted degree

4: **for** each vertex $v \in V$ **do**
 5: $wdeg[v] \leftarrow \sum_{u \in \mathcal{N}(v)} (1 - w_{vu})$

Step 4: Weighted k -Core decomposition (peeling)

6: $core[v] \leftarrow 0$ for all $v \in V$; $removed \leftarrow \emptyset$
 7: **while** $|removed| < |V|$ **do**
 8: $v^* \leftarrow \arg \min_{v \notin removed} wdeg[v]$ $\triangleright$ peel node with lowest degree
 9: $core[v^*] \leftarrow wdeg[v^*]$; $removed.append(\{v^*\})$
 10: **for** each $u \in \mathcal{N}(v^*)$, $u \notin removed$ **do**
 11: $wdeg[u] \leftarrow wdeg[u] - (1 - w_{v^*u})$

Step 5: Partition based on weighted core number

12: $V_D \leftarrow \{v \in V \mid core[v] \geq k^*\}$ $\triangleright$ dense, unstable core
 13: $V_A \leftarrow V \setminus V_D$ $\triangleright$ sparse, stable periphery

Step 6: Validation checks

14: **if** $|V_D| > 0.2 \cdot |V|$ **then**
 15: Increase k^* and repeat from Step 5 $\triangleright S$ too large, tighten threshold

Step 7: Build permutation

16: $P \leftarrow$ permutation matrix reordering $[V_A \mid V_D]$
 17: **return** (V_A, V_D, P)

iterations, where $T = 5 \cdot 2^{-\max\left(0, \frac{\delta F(X^K)}{F(X^K)} - \tau\right)}$. Here, τ defines the threshold below which the residual change is considered sufficiently small to safely maximize reuse. As the residual change decreases toward this threshold, T increases toward its maximum value of 5, allowing longer reuse intervals as the solver approaches convergence. Conversely, larger residual changes reduce T , prompting more frequent updates to maintain numerical accuracy.

B. Proposed Jacobian Partitioning for Schur Complement

As discussed earlier, the effectiveness of the Schur complement depends on partitioning the Jacobian matrix J so that stable, iteration-invariant regions are separated from dynamic, iteration-dependent ones. To avoid the high cost of explicit matrix reordering, we formulate this partitioning problem using a graph abstraction, in which each matrix index is represented as a vertex and each nonzero entry J_{ij} is a directed edge from vertex i to vertex j . The block partition (x_A, x_D) maps to vertex partitions, while the off-diagonal blocks B and C correspond to the edge cuts between vertex sets. Our objective is to partition vertices V into two disjoint subsets, V_A and V_D , such that three goals are achieved simultaneously: (1) the subgraph induced by V_A is sparse and dominated by iteration-invariant edges, enabling A to be safely frozen for T

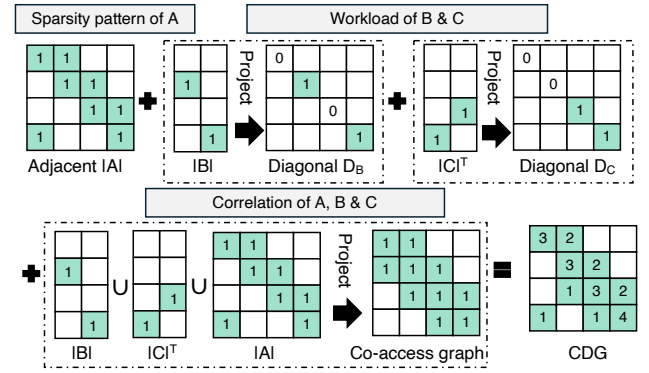


Fig. 6. An example of the proposed coupled dependency graph (CDG).

Newton iterations; (2) the cut between V_A and V_D is small, so that the off-diagonal blocks B and C remain sparse and inter-partition communication is minimized; and (3) $|V_D|$ is small relative to $|V|$, so that the resulting Schur complement S can fit on chip. Finding such a partition is a global combinatorial optimization problem and is NP-hard in general.

We observe that these objectives can be reformulated from the perspective of V_D . Specifically, the dynamic partition V_D should capture as many unstable edges as possible while remaining small in size, implying that it should form a compact, dense subgraph with iteration-dependent edges. In contrast, V_A corresponds to the sparse, iteration-invariant periphery of the graph. This aligns with the goal of k -core decomposition [47], which identifies densely connected vertices within a graph. However, standard k -core decomposition treats all edges uniformly and does not differentiate between stable and unstable edges, making it inadequate for our setting. To address this limitation, we propose a weighted k -core decomposition for Schur complement partition (WKCP), which incorporates edge stability as a weighting factor to bias the decomposition toward retaining vertices connected by unstable edges in V_D .

As detailed in Algorithm 2, Step 1 constructs a graph G from the adjacency matrix (i.e., Jacobian matrix). Step 2 assigns a binary stability weight to each edge: $w_{ij} = 1$ for stable edges, whose values change slowly across Newton iterations, and $w_{ij} = 0$ for unstable edges, whose values change iteratively. Step 3 computes the weighted degree of each vertex as the number of incident unstable edges, $wdeg[v] = \sum_{u \in \mathcal{N}(v)} (1 - w_{vu})$, which serves as the core score and reflects how deeply a vertex is embedded in the unstable-edge-rich region of the graph. Step 4 performs iterative peeling: at each round, the vertex with the smallest weighted degree is removed and assigned its current weighted degree as its core number, and the weighted degrees of its neighbors are decremented accordingly to reflect the removal of the peeled vertex's unstable-edge contributions. Step 5 partitions the vertices using a threshold k^* : vertices with core number greater than or equal to k^* are assigned to V_D , while the remaining vertices form V_A .

Figure 4 illustrates a walk-through example of the proposed WKCP. For simplicity, we use a symmetric matrix, though our methodology can be generalized to asymmetric sparsity patterns. Starting from the Jacobian graph with stable and

unstable edges annotated, the BFS traversal in the *Compute core score* stage computes the weighted degree of each vertex. Vertices with few unstable neighbors (e.g., V_0) receive low core scores, whereas vertices in the unstable-edge-rich region (e.g., V_3 – V_7) receive high scores. The peeling stage then proceeds over multiple rounds. In Rounds 1–3, V_0 , V_1 , and V_2 are successively peeled into V_A because they have the smallest weighted degrees, and the weighted degrees of their neighbors are updated after each removal. Rounds 4–5 continue the peeling process, with updated weighted degrees (e.g., $3 - 1 = 2$) reflecting the reduced unstable connectivity caused by earlier removals. The process terminates when all vertices have been assigned a core number, yielding a threshold of $k^* = 2$ that separates V_A from V_D . The resulting reordered matrix (rightmost panel) clearly exposes the block structure $[A; B; C; D]$, where A is sparse and stable, B and C are sparse blocks, and D is the dense, unstable block that is well suited for on-chip Schur complement computation.

V. PROPOSED TILING AND DATAFLOW

The dataflow and tiling optimization for the transformed nonlinear solver consists of two components: (1) Schur complement construction (i.e., $D - CA^{-1}B$), and (2) multiplying $(D - CA^{-1}B)$ by δx_D . Schur complement construction involves a sequence of matrix multiplications, while the remaining computations in the transformed nonlinear solver reduce to GEMV operations. GEMV has been well optimized and contains relatively limited room for data reuse optimization due to its low dimensionality. However, Schur complement construction introduces new challenges in tiling and dataflow mapping due to the tight dependencies among the involved matrices. Since A , B , C , and D are all decomposed from the same Jacobian matrix, they are inherently coupled rather than independent operands. This interdependence complicates both tile selection and dataflow design.

A. Proposed CDG-based Tiling Strategy for Schur Complement

Despite significant prior efforts [37], [48]–[51], existing works primarily focus on SpMMs where input data are independent. For example, I-GCN [49] clusters dense blocks into “islands” using BFS, while TeAAL [48] identifies tiles with the largest number of nonzeros to improve workload balance. However, simply reordering a single sparse matrix is insufficient for our problem, where matrices are not independent but instead exhibit strong data dependencies. In our setting, the partitioning of matrix A cannot be determined in isolation, because the submatrices B and C must be coordinated with the partitioning of A to enable efficient matrix multiplication and data reuse. For example, as shown in Figure 5, when computing $CA^{-1}B$, the row partition of submatrix C should be aligned with the column partition of submatrix B ; otherwise, additional communication is required for data exchange. Similar dependencies also incur when computing CA^{-1} and $A^{-1}B$ separately.

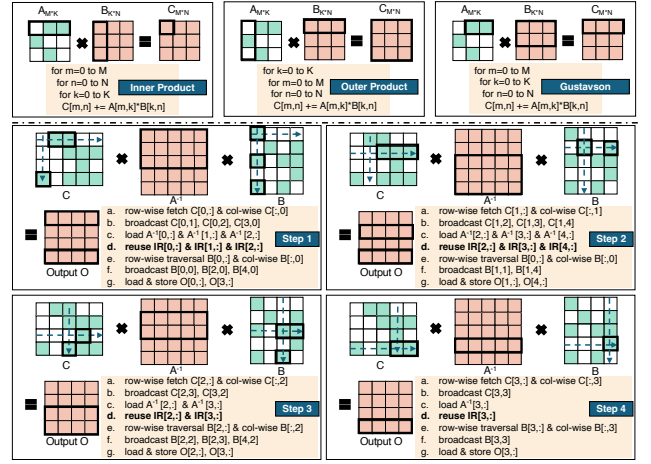


Fig. 7. Traditional dataflows and a walk-through example of the proposed Bi-index Generate-and-Scatter dataflow.

To capture the dependencies among the matrices A , B , and C , we formulate the problem as a graph partitioning problem over a unified dependency representation. Rather than partitioning each matrix independently, we develop a model that jointly captures both the sparsity structure and the inter-matrix dependencies introduced by Schur complement construction.

Specifically, we construct a new graph, called Coupled Dependency Graph (CDG), that incorporates the sparsity pattern of matrix A together with the workload and dependency information induced by B and C . In CDG, each vertex represents a column (or row) partition of matrix A along with the corresponding partition of dense matrix X_k . Edges derived from the sparsity pattern of A model the communication cost of sparse matrix multiplication, while additional vertex weights and dependency edges capture the workload contributions and data dependencies introduced by B and C .

Formally, we construct a weighted graph $CDG = (V, E, W)$ based on the matrix A . The CDG is defined as

$$CDG = |A| + D_B + D_C + G_{\text{dep}}, \quad (4)$$

where $|A| \in \mathbb{R}^{n \times n}$ represents the sparsity pattern of A (i.e., adjacent A). $D_B = \text{diag}(|B|1)$ and $D_C = \text{diag}(|C|^T 1)$ denote the projected workload contributions of matrices B and C onto the column space of A . The co-access graph $G_{\text{dep}} \in \mathbb{R}^{n \times n}$ captures the dependency structure, approximated by propagating the sparsity patterns of B and C through matrix A . Specifically, we can get G_{dep} by $(|AB(AB)^T|) \cup (|CA(CA)^T|)$. A walk-through example is shown in Figure 6. The CDG is built on three components: the adjacency matrix $|A|$ and the diagonal matrices D_B and D_C , which encode the number of nonzeros associated with each row or column. Consequently, the diagonal entries of the CDG represent the workload of each partition. The off-diagonal entries, in turn, capture both the sparsity pattern of A and the dependencies among A , B , and C .

The resulting Coupled Dependency Graph therefore captures three key factors simultaneously: (1) sparsity-induced communication from matrix A , (2) workload imbalance intro-

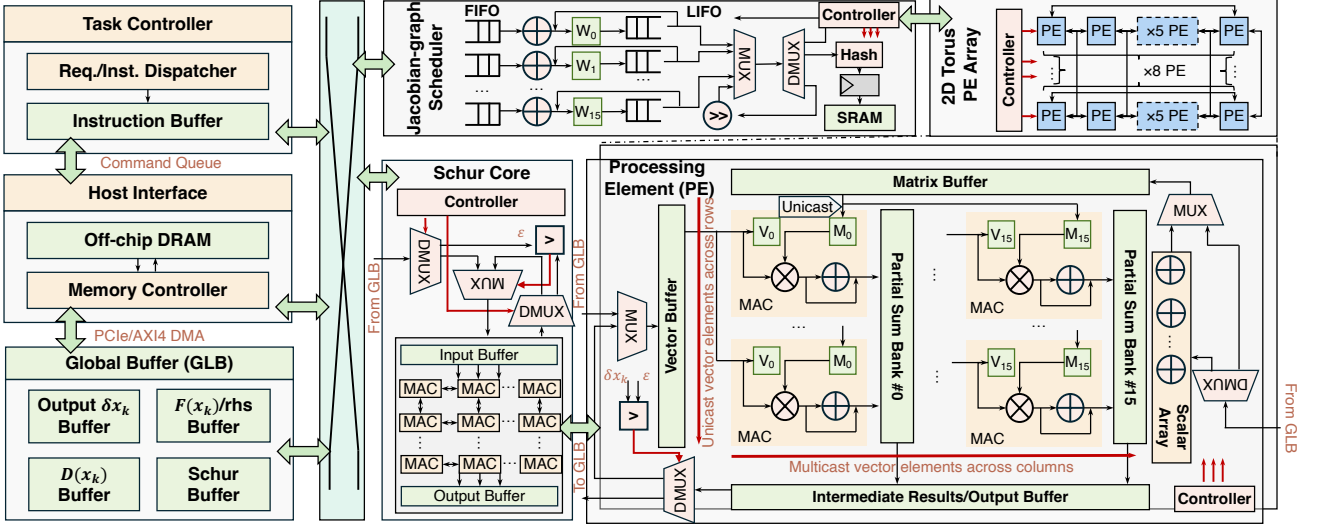


Fig. 8. The overview of the proposed architecture design.

duced by matrices B and C , and (3) inter-matrix dependencies arising from the Schur complement computation. Partitioning this graph allows us to jointly optimize data locality, workload balance, and communication overhead across all coupled matrices. With the CDG, we can leverage existing graph partitioning algorithms that aim to minimize edge cuts. In this work, we adopt METIS with the ϵ -imbalance parameter [52] to partition the CDG, as it has demonstrated strong scalability on large-scale graph partitioning tasks [53]–[55].

B. Proposed Bi-index Generate-and-Scatter Dataflow

A large set of dataflow optimizations has been studied to optimize data locality and parallelism for matrix multiplications. Three representative dataflows for SpMM [56]–[59] are illustrated in Figure 7. For example, inner-product dataflow, where a row $A[m, :]$ is multiplied with a column $B[:, n]$, optimizes the data locality for output C , but the data reuse for dense inputs is not entirely optimized. In contrast, outer-product dataflow pairs column $A[:, k]$ with row $B[k, :]$, maximizing output reuse at the cost of maintaining a large number of partial results. Gustavson dataflow multiplies all nonzero elements of a row in matrix A with corresponding rows in matrix B , which balances the reuse of both input and output matrices.

However, the primary bottleneck in chained matrix multiplication is frequent memory access to intermediate results (IR). To address this, we propose a Bi-index Generate-and-Scatter dataflow (BiGS) that maximizes the reuse of both inputs and intermediate results (IR). The key idea is to maintain a consistent dataflow between CA^{-1} and $CA^{-1}B$, ensuring that all generated IR are consumed promptly in the subsequent multiplication stage. Specifically, BiGS fetches the i -th row and i -th column of matrix C and multiplies them with the corresponding rows of matrix A^{-1} . This combines row-wise dataflow with an outer-product dataflow, maximizing the reuse of the dense matrix A^{-1} as well as partial sums. In addition, all generated intermediate results are immediately consumed

by the subsequent matrix multiplication to produce the final output.

For example, Figure 7 illustrates BiGS on a 9×9 original matrix. First, BiGS simultaneously fetches $C[0, :]$ and $C[:, 0]$. The nonzero entries $C[0, 1]$, $C[0, 2]$, and $C[3, 0]$ are broadcast, and the corresponding rows $A^{-1}[0, :]$, $A^{-1}[1, :]$, and $A^{-1}[2, :]$ are loaded. Critically, the generated intermediate results are scattered directly: $IR[0, :]$, $IR[1, :]$, and $IR[2, :]$ are combined with the nonzeros of $B[0, :]$ and $B[:, 0]$, which are broadcast as $B[0, 0]$, $B[2, 0]$, and $B[4, 0]$, to directly produce and store the final outputs $O[0, :]$ and $O[3, :]$. BiGS then advances to the next row and column, simultaneously fetching $C[1, :]$ and $C[:, 1]$. This bi-index access pattern maximizes both row reuse in A^{-1} and the reuse of intermediate results.

VI. PROPOSED ARGUS ACCELERATOR DESIGN

A. Architecture Overview

Although many accelerator architectures have been proposed for iterative solvers [1], [3], [10], most are highly specialized to particular application domains. Moreover, introducing Schur complement-based execution further increases the complexity of hardware design. To address these challenges, we propose **Argus**, a specialized accelerator architecture for sparse nonlinear solvers tailored to Schur complement-based computation.

Figure 8 shows the overall architecture of Argus. The design comprises five major components: a multi-bank Global Buffer (GLB), a Task Controller, a Jacobian-graph Scheduler, a Schur Core, and an array of Processing Elements (PEs) organized in an 8×8 2D torus topology. The accelerator interfaces with off-chip DRAM through a Memory Controller via PCIe/AXI4 DMA, while the PE array communicates with the GLB through a crossbar interconnect. The GLB is divided into four dedicated regions: an output δx_k buffer, an $F(x_k)$ /right-hand-side buffer, a $D(x_k)$ buffer, and a Schur buffer. Each region is associated with a distinct stage of the Schur complement-based nonlinear iterative solve.

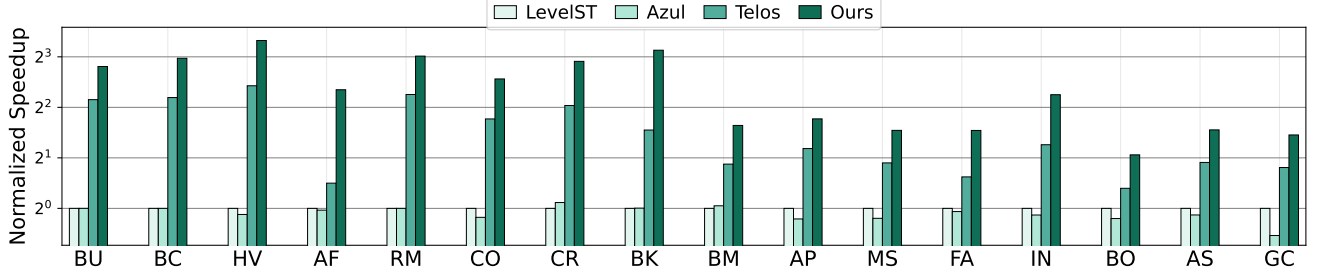


Fig. 9. Normalized speedup of baseline designs and Ours over LevelST across 16 datasets.

B. Proposed PE Microarchitecture

As illustrated in Figure 8, each Processing Element (PE) implements a unified architecture that simultaneously supports two computational phases: the iterative Schur complement solve $S\delta x_D = f_D - CA^{-1}f_A$ and the subsequent back-substitution, without requiring separate hardware units for each phase, improving resource utilization. Each PE comprises an Output Buffer, a Matrix Buffer, a Vector Buffer, a MAC Array, a Scalar Array, and 16 Partial Sum Banks. The local buffer size in each PE is 80 KB in total. Each PE contains a 16×16 MAC array, yielding 16,384 MACs in total. Inter-PE links are 96-bit wide with a latency of 1 cycle per hop.

The execution within each PE proceeds as follows. First, the sub-matrix $D(x_k)$ is loaded from the GLB into the PE at each Newton step, while the product $CA^{-1}B$, precomputed and cached by the Schur Core, is also fetched from the GLB. The Scalar Array then forms the Schur complement $S = D - CA^{-1}B$ in-place. Once S is available, the MAC Array executes the iterative GEMV operations: vector data is multicast along the row direction so that all MAC units within a row share the same vector element without redundant fetches, while matrix data is unicast along the column direction so that each MAC unit receives only its assigned matrix entry. Partial results from the MAC Array are accumulated into 16 banked Partial Sum Banks, where the 16-bank organization ensures that concurrent writes from different MAC units target distinct banks, eliminating write conflicts during parallel accumulation. Once all inner iterations converge (i.e., the control signal is produced by the comparator), the finalized δx_D is written back to the GLB output buffer, after which the back-substitution phase reuses the same MAC Array to compute δx_A by executing SpMV operations, amortizing the hardware cost across both solve phases.

C. Proposed Jacobian-graph Scheduler

The Jacobian-graph Scheduler implements the proposed WKCP algorithm in hardware to partition the Jacobian graph into (V_A, V_D) offline and dispatch the resulting sub-matrices to the Schur Core and PE Array, respectively. As illustrated in Figure 8, the scheduler processes the Jacobian graph metadata using a priority-queue-based approach [60]. Incoming adjacency and edge stability data from the GLB are buffered in multiple parallel input FIFOs, enabling concurrent ingestion of metadata across different vertex rows. Each FIFO feeds an adder ($\oplus$) that accumulates the number of incident unstable

edges for its assigned vertex, computing the weighted degree. The results are held in a 16-entry register file (W_0-W_{15}), which supports 16-way parallel weighted degree updates and reduces the $\mathcal{O}(|V|)$ sequential scan that a software implementation would require. An intermediate FIFO buffers the computed weighted degrees to smooth the data flow into the peeling stage. The MUX selects the vertex with the minimum weighted degree and pushes it onto a LIFO. The LIFO's stack discipline ensures that the most recently updated vertices are reconsidered first after each neighbor decrement, correctly implementing the iterative peeling procedure. The DMUX routes the vertex to either V_A or V_D accordingly. The Hash unit provides $\mathcal{O}(1)$ lookup of each vertex's current partition membership, and an SRAM (16 KB) stores the finalized partition (V_A, V_D) for consumption by the Schur Core and PE Array.

D. Proposed Schur Core

The Schur Core is responsible for computing A^{-1} and $CA^{-1}B$, whose results are retained on-chip to support inter-iteration reuse across T consecutive Newton steps. Rather than relying on the PE Array for these computations, we dedicate a separate hardware unit to decouple this phase from the PE Array's inner linear solve pipeline. A dedicated Schur Core allows the PE Array to continue executing the iterative solve without stalling to wait for Jacobian-dependent updates. Moreover, when recomputation of $CA^{-1}B$ is triggered, the Schur Core can perform the update in parallel with the PE Array's ongoing solve (the Schur Core updates $CA^{-1}B$ with δx_k and the PE Array continues calculating δx_{k+1}), ensuring sufficient pipeline parallelism and maximizing resource utilization across both units.

The Schur Core adopts an iterative matrix inversion strategy to compute A^{-1} , which avoids the fill-in overhead associated with direct factorization methods and preserves the sparse structure of A throughout computation. As illustrated in Figure 8, the Schur Core is built around a unified MAC array shared across both computational phases, with phase transitions managed through coordinated switching of the Input Buffer and Output Buffer contents via MUX/DMUX control logic. This unified design reduces hardware area while enabling direct on-chip transfer of A^{-1} from the first phase to the second without additional DRAM traffic.

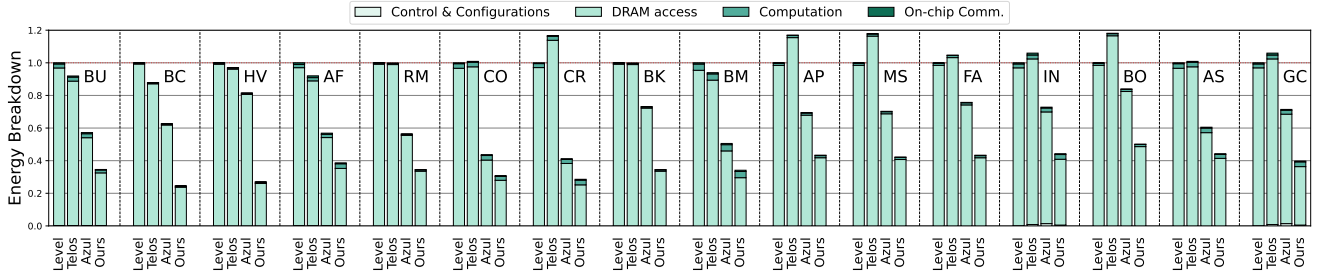


Fig. 10. Normalized energy consumption breakdown of baseline designs and Ours vs. LevelST across 16 datasets.

TABLE II
DIMENSIONS AND PROBLEM DOMAIN OF SUITESPARSE DATASETS.

Dataset	Abbr.	Rows	NNZ	Density	Problem Domain
l138_bus	BU	1,138	4,054	3.1×10^{-3}	Power Network Problem
bcsprw10	BC	5,300	21,842	7.8×10^{-4}	Power Network Problem
hvdcl	HV	24,842	158,426	2.6×10^{-4}	Power Network Problem
af23560	AF	23,560	460,598	8.3×10^{-4}	Computational Fluid Dynamics Problem
rma10	RM	46,835	2,329,092	1.1×10^{-3}	Computational Fluid Dynamics Problem
coater2	CO	9,540	207,308	2.3×10^{-3}	Computational Fluid Dynamics Problem
crankseg_1	CR	52,804	10,614,210	3.8×10^{-3}	Structural Problem
bcsstk18	BK	11,948	149,090	1.0×10^{-3}	Structural Problem
bcsstm25	BM	15,439	15,439	6.5×10^{-5}	Structural Problem
apache2	AP	715,176	4,817,870	9.4×10^{-6}	Structural Problem
msdoor	MS	415,863	19,173,163	1.1×10^{-4}	Structural Problem
Fault_639	FA	638,802	27,245,944	6.7×10^{-5}	Structural Problem
inline_1	IN	503,712	36,816,170	1.5×10^{-4}	Structural Problem
boneS01	BO	127,224	5,516,602	3.4×10^{-4}	Model Reduction Problem
af_shell8	AS	504,225	17,579,155	6.9×10^{-5}	Subsequent Structural Problem
G3_circuit	GC	1,585,478	7,660,826	3.0×10^{-6}	Circuit Simulation Problem

VII. EVALUATION

A. Evaluation Setup

Configurations. We build a cycle-accurate simulator for Argus, following methodologies from prior works [1], [2], [37], [39]. The simulator integrates with Ramulator [61] to model HBM [62] with 256 GB/s bandwidth and captures the cycle-level behavior of all compute and memory components. Argus comprises 64 PEs organized in an 8×8 2D torus topology, each with a 16×16 FP32 MAC array (16,384 MACs total), operating at 1 GHz. The 64 MB global buffer (GLB) is partitioned into four dedicated regions: output δx_k buffer (4 MB), $F(x_k)/r_{\text{rhs}}$ buffer (4 MB), $D(x_k)$ buffer (16 MB), and Schur buffer (40 MB). Local PE buffers are 80 KB each. Inter-PE links are 96-bit wide with 1-cycle per-hop latency. The Schur Core contains an independent 16×16 MAC array with dedicated local buffers (64 KB). For ASIC evaluation, we synthesize in Verilog using TSMC 32nm with Synopsys Design Compiler. Switching activity is captured via waveform traces and analyzed using PrimeTime PX. On-chip buffer energy and area are modeled with CACTI 7.0 [63]; the NoC energy is modeled using DSENT [64] provided in BookSim2 [65]. For GPU experiments, we use a server equipped with an Intel® Xeon® Platinum 8362 CPU @ 2.80GHz and an NVIDIA A100 80GB GPU [66].

Datasets. We evaluate Argus using sixteen sparse datasets to represent Jacobian, spanning power network problems (BU, BC, HV), computational fluid dynamics (AF, RM, CO), structural problems (CR, BK, BM, AP, MS, FA, IN), model reduction (BO), subsequent structural problems (AS), and

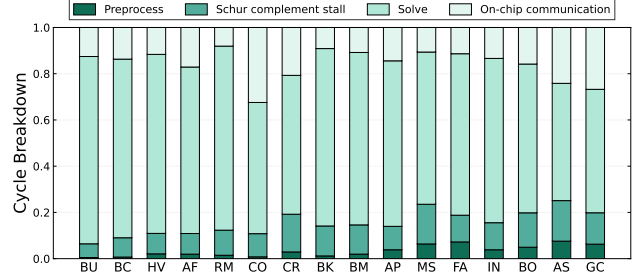


Fig. 11. End-to-end cycle breakdown of Argus's PE.

circuit simulation (GC) from SuiteSparse [67], which are representative of sparsity patterns in real-world problems. These datasets vary in dimension (1,138 to 1,585,478 rows), nonzero count (4,054 to 36,816,170 NNZs), and density (3.0×10^{-6} to 3.8×10^{-3}), providing a comprehensive benchmark suite for sparse nonlinear systems. The Jacobian matrices exhibit diverse sparsity structures: symmetric matrices from structural mechanics and fluid dynamics, nonsymmetric matrices from power flow and circuit simulation, and both slowly-varying (structural) and rapidly-varying (circuit) numerical patterns across Newton iterations. Dataset properties are summarized in Table II.

Baseline Platforms. We evaluate Argus against three state-of-the-art iterative linear solver accelerators—Azul [1], LevelST [2], and Telos [3]. To ensure fairness, we adapt the baseline architectures to the nonlinear setting by executing the full Newton-PCG algorithm (Algorithm 1) on each platform, using five Newton iterations with five inner linear iterations per Newton step. For Azul, we replicate its hypergraph-based data mapping and on-chip SRAM retention strategy, but trigger Jacobian reload at every outer Newton iteration. Specifically, it formulates the sparse matrices and vectors using a hypergraph. We implement the hypergraph partitioning by employing PaToH v3.3 [68]. For LevelST, we implement its stream-based SpTRSV execution with sequential data tiles for the preconditioner solve. For Telos, we replicate its plane-parallel pipelining technique for structured sparsity. All designs are scaled to match Argus's total MAC count (16,384 MACs), organized into 64 PEs at 1 GHz. The GLB size is uniformly set to 64 MB. For a comprehensive comparison, Argus's evaluation incorporates the Schur complement computation overhead (A^{-1} , $CA^{-1}B$ computation), kernel execution across all Newton iterations, and DRAM access time cycles.

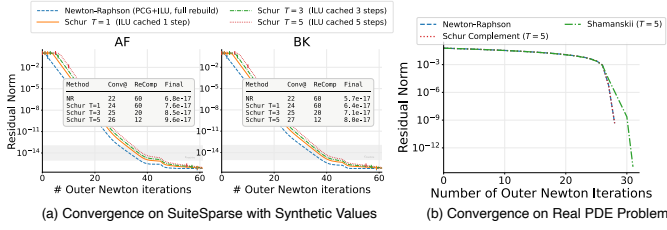


Fig. 12. Convergence analysis of the proposed Schur complement reformulation.

B. Performance Analysis

We evaluate the normalized speedup of Argus and baselines relative to LevelST across 16 datasets. As shown in Figure 9, Argus achieves geometric mean speedups of $1.86\times$, $5.32\times$, and $5.68\times$ over Azul, LevelST, and Telos, respectively. Argus outperforms all baselines for fundamentally different reasons rooted in their inability to handle dynamic Jacobian updates in nonlinear solvers. LevelST adopts a stream-based SpTRSV model assuming a fixed coefficient matrix, forcing full Jacobian reloads at each Newton iteration and incurring repeated off-chip accesses. Azul mitigates communication via hypergraph-based mapping to retain the matrix on-chip across iterations, but assumes fixed sparsity and cannot handle dynamic updates efficiently. Telos exploits structured sparsity via plane-parallel pipelining, achieving high efficiency on stencil-based PDEs but degrading on unstructured nonlinear systems due to irregular sparsity, which limits PE utilization and parallelism. Argus addresses these limitations through a unified PE architecture supporting compact GEMV patterns, combined with CDG-based tiling and BiGS dataflow to maintain workload balance and minimal communication regardless of sparsity structure. Performance gains vary across datasets with different sparsity characteristics. For moderately sized and denser datasets such as RM and CR, Argus achieves the largest speedups. Here, WKCP produces well-structured block decompositions where V_D remains compact while A retains sufficient density to sustain high MAC utilization during iterative inversion. The resulting Schur complement S is small and dense, enabling effective intra-iteration reuse. In contrast, extremely sparse or large datasets show smaller gains. On GC, extreme sparsity leads to highly irregular nonzero distributions, reducing per-PE utilization and limiting throughput gains.

C. Energy Efficiency Analysis

To assess the energy consumption, we compare the energy breakdown of Argus against LevelST, Telos, and Azul, as shown in Figure 10. Overall, Argus reduces energy consumption by 62.82%, 63.93%, and 42.06% over LevelST, Telos, and Azul, respectively. The dominant source of energy reduction is off-chip DRAM access, which constitutes the largest energy component across all three baselines due to their inability to exploit inter-iteration reuse. Azul partially mitigates this through hypergraph-based scheduling, but still suffers from full Jacobian reloads triggered by dynamic updates, leading to substantial DRAM energy overhead. In contrast, Argus substantially reduces the DRAM access energy component

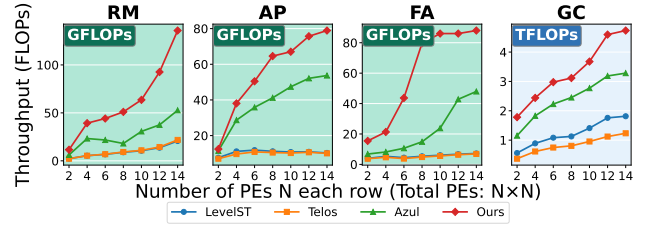


Fig. 13. Throughput across varying numbers of PEs.

that dominates baseline execution through the proposed Schur complement for nonlinear solvers, achieving a more favorable energy distribution across all datasets. On GC, the extremely low density and large matrix dimension result in highly irregular nonzero distributions, where the DRAM access reduction from inter-iteration reuse is comparatively smaller, as the sparse Jacobian graph produces a less compact V_D and thus a larger residual DRAM footprint per Newton step.

D. PE Cycle Breakdown

Figure 11 shows the breakdown of cycles spent in the proposed Argus PEs. We decompose total PE cycles into four components: preprocessing, Schur complement stall, solve, and on-chip communication. On average, the solve phase accounts for 64.76% of total execution time, while the Schur complement stall constitutes only 13.86%, and on-chip communication and preprocessing together contribute the remaining 21.38%. The low preprocessing overhead reflects the efficiency of the hardware-implemented WKCP algorithm in the Jacobian-graph Scheduler. The preprocessing cost is thus amortized efficiently across Newton iterations. The minimal Schur complement stall validates the pipeline parallelism design of the Schur Core: by decoupling $CA^{-1}B$ computation from the PE Array's inner solve, the Schur Core can update $CA^{-1}B$ in parallel with the ongoing δx_{k+1} computation, hiding recomputation latency entirely from the critical path. The dominance of the solve phase indicates that Argus's intra-iteration reuse is functioning as intended; the PE Array sustains high MAC utilization without off-chip memory stalls.

In comparison, as shown in Figure 10, DRAM access constitutes the dominant energy component for all three baselines, directly reflecting their inability to exploit either inter- or intra-iteration reuse; Argus substantially reduces this component by keeping the working set on-chip. The irregular sparsity structure of GC slightly increases the Schur complement stall relative to other datasets, as the iterative matrix inversion [69] requires more iterations to converge on the denser and less structured sub-matrix.

E. Convergence Analysis

To evaluate the numerical convergence performance of Newton-Raphson with Schur complement, we construct synthetic nonlinear functions $F(u) = \text{diag}(1 + \alpha u^2) \cdot J_{\text{pattern}} \cdot u - b = 0$ whose sparsity patterns are generated via preferential-attachment graphs matching the statistical characteristics of two real-world SuiteSparse matrices. We compare standard Newton-Raphson with PCG+ILU(0) against our Schur complement variant at recompute intervals $T \in \{1, 3, 5\}$, running

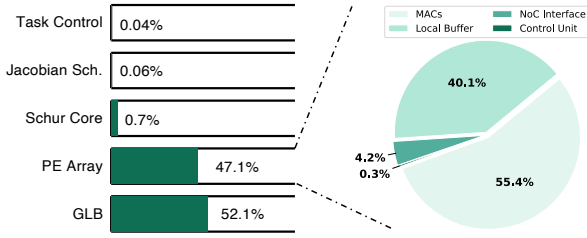


Fig. 14. Area breakdown of Argus Accelerator.

all methods for a maximal budget of 60 outer Newton iterations with backtracking line search to ensure a fair comparison. As shown in Figure 12(a), the use of the Schur complement and persistently storing the sub-matrices introduces a mild convergence slowdown. These results confirm that the proposed Schur complement formulation trades a negligible convergence overhead for a significant reduction in the most expensive operation of the nonlinear solve, validating the practical efficiency of inter-iteration reuse across a range of recompute intervals.

Real-world Non-linear PDE Problem. To complement the synthetic convergence study in Figure 12(a), we evaluate the proposed Schur complement formulation on PETSc’s Bratu benchmark (src/snes/tutorials/ex5) [70], [71] on an A100 GPU. A custom Newton solver selectively freezes the stable off-diagonal Jacobian entries while refreshing the unstable diagonal entries, whereas PETSc’s `-snes_lag_jacobian` option freezes the entire Jacobian, corresponding to the classical Shamanskii method [72]. As shown in Figure 12(b), both Newton–Raphson and the proposed method converge in 28 outer iterations, while Shamanskii requires 31 iterations to reach the same residual floor. Unlike Figure 12(a), the residual remains nearly unchanged for more than 20 iterations before dropping rapidly near iteration 25. This slower initial convergence is caused by PETSc’s damped Newton update, which applies a step-size damping parameter to reduce the Newton step in the early iterations. While this improves robustness by preventing overly aggressive updates, it also delays entry into the local quadratic-convergence regime. In contrast, the undamped Newton solve in Figure 12(a) enters this regime much earlier, leading to a steadily accelerating residual reduction.

F. Throughput Analysis

We evaluate throughput on Torus topologies from 2×2 to 14×14 , covering small and large scales. We record simulation cycles and total data bytes. As shown in Figure 13, Argus achieves $6.29\times$, $6.87\times$, and $2.31\times$ higher throughput than LevelST, Telos, and Azul, respectively. These gains stem from each baseline’s limited scalability with increasing compute resources. LevelST is constrained by the sequential dependency of SpTRSV, which limits parallelism and causes throughput to plateau at large PE counts; its chunk-based tiling also ignores unstructured sparsity, leading to workload imbalance and growing communication overhead. Argus avoids these issues by decomposing the problem into parallel-friendly GEMV, while CDG-based partitioning ensures workload balance and

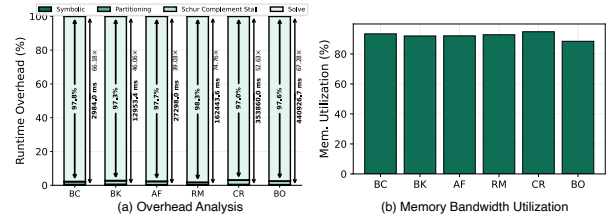


Fig. 15. (a) Overhead analysis. (b) Memory bandwidth utilization.

minimal inter-tile communication. Azul’s hypergraph mapping incurs substantial cross-PE communication during Jacobian reloads, which grows with PE count. Telos relies on structured sparsity, resulting in low PE utilization on unstructured inputs as more PEs remain idle. In contrast, Argus’s unified PE architecture adapts to mixed computation patterns, sustaining high utilization as PE count scales.

G. Overhead Analysis

Area Breakdown. Figure 14 presents the area breakdown of the Argus accelerator. At the system level, GLB accounts for 52.1% of the total area, while the PE array occupies 47.1%. Within each PE, 40.1% of the area is devoted to local buffers and registers, whereas 55.4% is allocated to MAC units. These proportions reflect a design choice that emphasizes substantial per-PE local storage, enabling the proposed dataflow to effectively exploit intra-iteration reuse while minimizing frequent accesses to the GLB. The Jacobian-graph scheduler incurs only 0.06% of the total area overhead.

Symbolic Overhead Analysis. To quantify the overhead introduced by identifying stable and unstable Jacobian entries, we profile the overhead analysis across six datasets on an A100 GPU. We count 10 Newton iterations for all datasets. As shown in Figure 15(a), the combined symbolic analysis, partitioning overhead, and Schur complement stall account for only 2.4% of total runtime on average. This trend holds consistently regardless of matrix dimension or density. These results confirm that the stability classification is a lightweight, one-time preprocessing step whose cost is effectively amortized across the Newton iterations.

GPU Comparison. We further compare the performance speedup of Argus over the GPU implementation. Figure 15(a) shows Argus outperforms GPU by $57.66\times$, indicating the necessity to design an ASIC accelerator for nonlinear solvers.

Memory Bandwidth Utilization. Figure 15(b) indicates Argus achieves consistently high memory bandwidth utilization, achieving up to 94.83% of the peak HBM bandwidth. This indicates that Argus’s Schur complement reformulation, combined with CDG-based tiling and BiGS dataflow, keeps the design close to memory-bound operation rather than leaving bandwidth idle, with BO showing the lowest utilization due to its extreme sparsity.

H. Performance Breakdown

To attribute the end-to-end speedup to its constituent architectural components, we decompose Argus into five cumulative configurations and evaluate each using an analytical cost model grounded in real SuiteSparse matrix statistics: (A) the

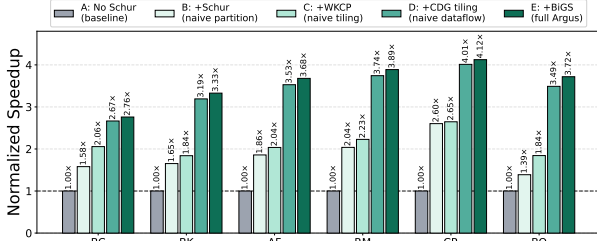


Fig. 16. Performance breakdown across datasets.

baseline Newton-PCG solver without the Schur complement reformulation; (B) Schur complement reformulation with a naive contiguous partition in place of WKCP; (C) WKCP partitioning with naive, dependency-unaware tiling in place of CDG; (D) CDG-based tiling on top of WKCP, without the BiGS dataflow; and (E) the full Argus design with BiGS dataflow. As shown in Figure 16, Argus achieves cumulative speedups of $2.76\times$ – $4.12\times$ over the no-Schur baseline. The Schur complement reformulation alone (Config. B) already yields $1.39\times$ – $2.60\times$ speedup by converting the dominant SpMV pattern into a dense GEMV that better utilizes the PE array’s MAC throughput, confirming that the reformulation contributes the most significant performance gains. Replacing the naive partition with WKCP (Config. C) contributes a further but comparatively modest $1.02\times$ – $1.32\times$ increment on most datasets, because the partition-quality penalty in our model only manifests when a substantial fraction of unstable edges falls outside V_D ; for the matrices considered, the naive partition is already large enough (20–27% of V_D , matched to WKCP’s output size) to capture a non-trivial share of the unstable structure by chance. The largest secondary contribution comes from CDG-based tiling (Config. D), which yields $1.30\times$ – $1.90\times$ additional speedup over Config. C by reducing the communication penalty associated with cut edges between the A , B , and C submatrices; this is consistent with Section IV-A’s motivation that naive tiling, which ignores the inter-matrix dependencies introduced by the Schur complement construction, leaves significant cross-tile communication unaddressed. BiGS dataflow (Config. E) contributes the smallest increment, typically 3%–7% over Config. D. Because on most of these datasets, the bottleneck has already shifted from on-chip compute to DRAM bandwidth by the time CDG tiling is applied, further compute-side optimization yields diminishing end-to-end returns once memory traffic dominates.

I. Scalability Study

Figure 17 presents the normalized speedup of Argus as PE count scales from 8 to 512 across four representative datasets. The results show that scalability is closely related to matrix density. AP exhibits the strongest speedup improvement as the PE count increases, reaching $26\times$ speedup at 512 PEs. This is because AP has a large matrix dimension but extremely low density. At small PE counts, the sparse workload is too large and irregular to be distributed efficiently, whereas additional PEs enable finer-grained partitioning across tiles and expose more parallelism. In contrast, RM, which has a moderate

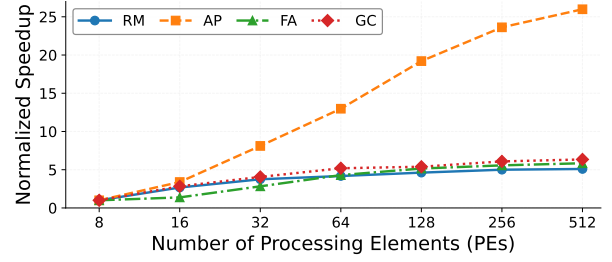


Fig. 17. Normalized speedup for varying number of PEs.

matrix size and relatively higher density, achieves more stable and near-linear scaling across all PE configurations.

VIII. CONCLUSION

In conclusion, we posit that decoupling the static components from the dynamic Jacobian matrix, rather than recomputing the entire matrix at every iteration, can effectively exploit both data reuse and parallelism. Based on this insight, we propose Argus, a specialized accelerator architecture built on a Schur complement-based reformulation of the Newton-Raphson method. Specifically, we reformulate Jacobian computation through the Schur complement to decouple static and dynamic sparsity patterns from the Newton-Raphson method, reducing the reload and recomputation of the Jacobian matrix over iterations. In addition, the reformulation also converts sparse matrix-vector multiplication (SpMV) of nonlinear computations to general matrix-vector multiplication (GEMV). We further propose a modified weighted k -core decomposition method and construct a new Coupled Dependency Graph to avoid communication bottlenecks while improving the data locality for Schur complement computations. Lastly, we design a novel accelerator architecture and dataflow mapping that can support both Schur complement and GEMV computations. Our evaluation shows that Argus delivers performance speedups of $1.86\times$, $5.32\times$, and $5.68\times$ over state-of-the-art iterative solver designs Azul [1], LevelST [2], and Telos [3], respectively.

ACKNOWLEDGMENT

This work was partially supported by the U.S. National Science Foundation under CAREER Award CCF-2441973 and Award CCF-2551105, and the U.S. Department of Energy under grant DE-SC0024428. We sincerely thank the reviewers for their valuable comments.

REFERENCES

- [1] Axel Feldmann, Courtney Golden, Yifan Yang, Joel S Emer, and Daniel Sanchez. Azul: An accelerator for sparse iterative solvers leveraging distributed on-chip memory. In *Proceedings of 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 643–656. IEEE, 2024.
- [2] Zifan He, Linghao Song, Robert F Lucas, and Jason Cong. Levelst: Stream-based accelerator for sparse triangular solver. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 67–77, 2024.
- [3] Xiaochen Hao, Hao Luo, Chu Wang, Chao Yang, and Yun Liang. Telos: A dataflow accelerator for sparse triangular solver of partial differential equations. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1792–1805, 2025.
- [4] Philip Rabinowitz et al. Numerical methods for nonlinear algebraic equations. 1970.

- [5] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 31, 2018.
- [6] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Deep equilibrium models. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.
- [7] Tjalling J Ypma. Historical development of the newton–raphson method. *SIAM review*, 37(4):531–551, 1995.
- [8] Ron S Dembo, Stanley C Eisenstat, and Trond Steihaug. Inexact newton methods. *SIAM Journal on Numerical Analysis*, 19(2):400–408, 1982.
- [9] Dana A Knoll and David E Keyes. Jacobian-free newton–krylov methods: a survey of approaches and applications. *Journal of Computational Physics (JCP)*, 193(2):357–397, 2004.
- [10] Jiajun Li, Yuxuan Zhang, Hao Zheng, and Ke Wang. Fdmax: An elastic accelerator architecture for solving partial differential equations. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2023.
- [11] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *Proceedings of 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 249–260. IEEE, 2020.
- [12] Jiajun Li and Yue Deng. Spadix: A highly efficient accelerator for solving 3-d partial differential equations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 43(9):2797–2809, 2024.
- [13] Ubaid Bakhtiar, Helya Hosseini, and Bahar Asgari. Acamar: A dynamically reconfigurable scientific computing accelerator for robust convergence and minimal resource underutilization. In *Proceedings of 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1601–1616. IEEE, 2024.
- [14] David E Keyes, Lois C McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, et al. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications (IJHPCA)*, 27(1):4–83, 2013.
- [15] Jiri Vlach and Kishore Singhal. *Computer methods for circuit analysis and design*. Springer Science & Business Media, 1983.
- [16] Daniele Mortari. Esoq: A closed-form solution to the wahba problem. *The Journal of the Astronautical Sciences*, 45(2):195–204, 1997.
- [17] Stanley C Eisenstat. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM Journal on Scientific and Statistical Computing (SISC)*, 2(1):1–4, 1981.
- [18] Mayez Al-Mouhamed, Lutfi Firdaus, Ayaz H Khan, and Nazeeruddin Mohammad. Spmv and bicg-stab sparse solver on multi-gpus for reservoir simulation. *Multimedia Tools and Applications*, 83(8):23563–23597, 2024.
- [19] Gerard LG Sleijpen, Henk A Van der Vorst, and Diederik R Fokkema. Bicgstab (l) and other hybrid bi-cg methods. *Numerical Algorithms*, 7(1):75–109, 1994.
- [20] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing (SISC)*, 7(3):856–869, 1986.
- [21] Michael Allen Heroux, Jack Dongarra, and Piotr Luszczek. Hpec benchmark technical specification. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2013.
- [22] Ya-xiang Yuan. A modified bfgs algorithm for unconstrained optimization. *IMA Journal of Numerical Analysis*, 11(3):325–332, 1991.
- [23] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.
- [24] Manoj B Rajashekar, Xingyu Tian, and Zhenman Fang. Hismv: Hybrid row distribution and vector buffering for imbalanced spmv acceleration on fpgas. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 154–164, 2024.
- [25] Guoqing Xiao, Kenli Li, Yuedan Chen, Wangquan He, Albert Y Zomaya, and Tao Li. Caspmv: A customized and accelerative spmv framework for the sunway taihulight. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 32(1):131–146, 2019.
- [26] Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari. Chason: Supporting cross hbm channel data migration to enable efficient sparse algebraic acceleration. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 778–794, 2025.
- [27] Abdul Rehman Tareen, Marius Meyer, Christian Plessl, and Tobias Kenter. Hihismv: Sparse matrix-vector multiplication with hierarchical row reductions on fpgas with high-bandwidth memory. In *Proceedings of 2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 32–42. IEEE, 2024.
- [28] Courtney Golden, Axel Feldmann, Joel Emer, and Daniel Sanchez. Quartz: A reconfigurable, distributed-memory accelerator for sparse applications. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 929–943, 2025.
- [29] Helya Hosseini, Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. Segin: Synergistically enabling fine-grained multi-tenant and resource-optimized spmv. *IEEE Computer Architecture Letters (CAL)*, 2025.
- [30] Fuzhen Zhang. *The Schur complement and its applications*. Springer Science & Business Media, 2006.
- [31] Sergey Nepomnyashchikh. Domain decomposition methods. *Radon Series Comput. Appl. Math.*, 1:89–159, 2007.
- [32] Jeffrey P Van Doormaal and George D Raithby. Enhancements of the simple method for predicting incompressible fluid flows. *Numerical heat transfer*, 7(2):147–163, 1984.
- [33] Neema Jamshidi and Bernhard Ø Palsson. Mass action stoichiometric simulation models: incorporating kinetics and regulation into stoichiometric models. *Biophysical journal*, 98(2):175–185, 2010.
- [34] Wolfram Liebermeister and Edda Klipp. Bringing metabolic networks to life: convenience rate law and thermodynamic constraints. *Theoretical Biology and Medical Modelling*, 3(1):41, 2006.
- [35] Xiaowei Jin, Shengze Cai, Hui Li, and George Em Karniadakis. Nsfnets (navier-stokes flow nets): Physics-informed neural networks for the incompressible navier-stokes equations. *Journal of Computational Physics (JCP)*, 426:109951, 2021.
- [36] Sanjali Yadav, Amirmahdi Namjoo, and Bahar Asgari. Misam: Machine learning assisted dataflow selection in accelerators for sparse matrix multiplication. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 824–838, 2025.
- [37] Yifan Yang, Joel S Emer, and Daniel Sanchez. Trapezoid: A versatile accelerator for dense and sparse matrix multiplications. In *Proceedings of 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 931–945. IEEE, 2024.
- [38] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. Spade: A flexible and scalable accelerator for spmm and sddmm. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 1–15. IEEE, 2023.
- [39] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 747–761. ACM, 2023.
- [40] Amir Ghazizadeh Ahsaei, Lingxiang Yin, Shilin Tian, Fangzhou Ye, Fan Yao, and Hao Zheng. Rethinking tiling and dataflow for spmm acceleration: A graph transformation framework. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1535–1548, 2025.
- [41] Linghao Song, Yuze Chi, Atefeh Sohrabzadeh, Young-kyu Choi, Jason Lau, and Jason Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 65–77, 2022.
- [42] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraport: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proceedings of 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE, 2020.
- [43] Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. Pipirima: Predicting patterns in sparsity to accelerate matrix algebra. In *Proceedings of 2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7, 2025.
- [44] Fangzhou Ye, Shilin Tian, Amir Ghazizadeh Ahsaei, and Hao Zheng. Tensorprism: Rethinking sparse high-order tensor acceleration via co-occurrence graph. In *Proceedings of 2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*, pages 291–306. IEEE, 2026.

- [45] Axel Feldmann and Daniel Sanchez. Spatula: A hardware accelerator for sparse matrix factorization. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 91–104, 2023.
- [46] M Necati Özişik, Helcio RB Orlande, Marcelo J Colaço, and Renato M Cotta. *Finite difference methods in heat transfer*. CRC press, 2017.
- [47] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on principles of distributed computing (PODC)*, pages 207–208, 2011.
- [48] Nandeeka Nayak, Toluwanimi O Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. Teaal: A declarative framework for modeling sparse tensor accelerators. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1255–1270, 2023.
- [49] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1051–1063. IEEE, 2021.
- [50] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNNs acceleration on GPUs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 515–531. USENIX, 2021.
- [51] Fangzhou Ye, Lingxiang Yin, Amir Ghazizadeh Ahsaei, and Hao Zheng. Egma: Enhancing data reuse and workload balancing in message passing gnn acceleration via gram matrix optimization. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2024.
- [52] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.
- [53] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 103–117, 2023.
- [54] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.
- [55] Fangzhou Ye, Lingxiang Yin, and Hao Zheng. Scaling graph neural network training via geometric optimization. In *Proceedings of 2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–15. IEEE, 2026.
- [56] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–333. IEEE, 2019.
- [57] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.
- [58] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. GROW: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 42–55. IEEE, 2023.
- [59] Eric Qin, Ananda Samajdar, Hyounkjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *Proceedings of 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.
- [60] Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, and Lingling Tong. Priority queues and dijkstra’s algorithm. 2007.
- [61] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters (CAL)*, 15(1):45–49, 2016.
- [62] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm dram technology and architecture. In *Proceedings of IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2017.
- [63] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.
- [64] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. Dsent-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip (NOCS)*, pages 201–210. IEEE, 2012.
- [65] Bheemappa Halavar, Ujjwal Pasupulety, and Basavaraj Talawar. Extending booksim2. 0 and hotspot6. 0 for power, performance and thermal evaluation of 3d noc architectures. *Simulation Modelling Practice and Theory*, 96:101929, 2019.
- [66] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.
- [67] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.
- [68] Ümit V Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs)., 2011.
- [69] Victor Pan and Robert Schreiber. An improved newton iteration for the generalized inverse of a matrix, with applications. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1109–1130, 1991.
- [70] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William Gropp, et al. Petsc users manual. 2019.
- [71] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. *Modern Software Tools in Scientific Computing*, pages 163–202, 1997.
- [72] VE Shamanskii. On the application of newton’s method in a singular case. *USSR Computational Mathematics and Mathematical Physics*, 7(4):72–85, 1967.