SPIKe: Generating Practical SpMV Kernels for Sparse Iterative Methods on GPUs

Luhan Wang
School of Computer Science
Peking University
Beijing, China
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
wlh@stu.pku.edu.cn

Haipeng Jia*
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
jiahaipeng@ict.ac.cn

Kun Li
Institute for AI Industry Research
Tsinghua University
Beijing, China
likun@air.tsinghua.edu.cn

Zongyuan He
School of Computer Science
Peking University
Beijing, China
zyhe@pku.edu.cn

Shengguo Li
National University of Defense Technology
Changsha, China
shengguolsg@126.com

Ting Cao
Institute for AI Industry Research
Tsinghua University
Beijing, China
tingcao@mail.tsinghua.edu.cn

Yunxin Liu
Institute for AI Industry Research
Tsinghua University
Beijing, China
liuyunxin@air.tsinghua.edu.cn

Yunquan Zhang
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
zyq@ict.ac.cn

Yifeng Chen
School of Computer Science
Peking University
Beijing, China
cyf@pku.edu.cn

***Abstract*—Sparse iterative methods with static sparse matrices form the backbone of many applications, where SpMV is the primary performance bottleneck. Conventional optimizations typically involve costly matrix-specific preprocessing, causing unamortizable overhead as algorithmic advances reduce iteration counts. We present SPIKe, a practical SpMV kernel generator targeting efficient sparse iterative methods across a wide spectrum of iteration counts. SPIKe comprises three key techniques: 1) Online *Thread-Level Parallelism Adaptation* policy, using a thread-computed row span proxy to enhance load balance for uniform tiles; 2) Online *Warp-Level Asymmetric Scheduling* policy, featuring a lightweight Warp-as-a-Sensor to route skewed tiles to asymmetric code paths; 3) Offline *Architecture-Domain Co-Aware Code Generation* for kernel optimization following the preceding policies. Across 2,294 matrices, SPIKe outperforms six baselines with negligible preprocessing overhead. In end-to-end evaluations spanning varying iteration counts, SPIKe achieves average speedups of 2.78x for PageRank and 4.26x for GMRES against GraphBLAS and PETSc.**

***Index Terms*—Sparse Iterative Method, Preprocessing, SpMV**

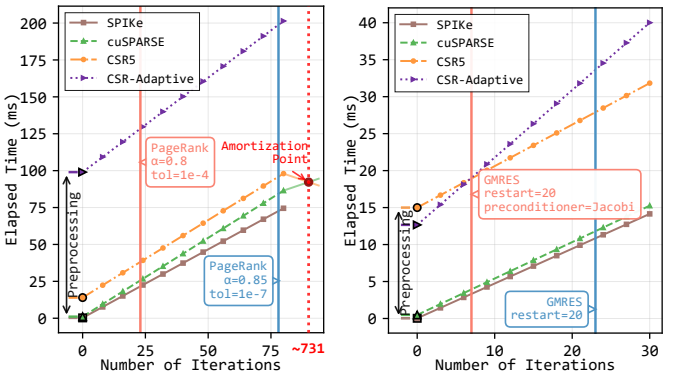
I. Introduction

Sparse iterative methods form the computational backbone of numerous applications, from linear solvers [1] to graph analytics [2]. The performance of these iterative methods heavily relies on Sparse Matrix-Vector multiplication (SpMV), which is invoked repeatedly during iterations. However, the irregular non-zero distribution of sparse matrices often leads to poor memory access locality [3] and load imbalance on GPUs [4]. Because sparse matrices typically remain static across iterations, researchers often develop sophisticated matrix-specific preprocessing techniques [4]–[7] to regularize non-zero distributions and accelerate the SpMV kernel.

Yet, these techniques are not practical in sparse iterative methods with a wide spectrum of iteration counts. When considering total time-to-solution, seemingly high-performance kernels can become uncompetitive due to heavy online preprocessing costs. Fig. 1 shows that PageRank using the power method converges in only 23 iterations on the europe_osm matrix, while generalized minimal residual method (GMRES) with Jacobi preconditioning converges in just 7 iterations on cage15. Although the well-known CSR5 [5] shows marginally superior kernel performance on europe_osm, it does not outperform cuSPARSE [8] until ∼731 iterations (Fig. 1(a)). This indicates that its preprocessing overhead prevents it from achieving superior end-to-end performance within practical convergence iteration counts. Furthermore, preprocessing-based implementations sometimes exhibit inferior kernel performance (Fig. 1(b)), negating potential advantage. This reveals a fundamental contradiction in the preprocessing-centric paradigm: ***achieving higher kernel performance demands increasingly sophisticated preprocessing,***

*Corresponding author (jiahaipeng@ict.ac.cn).

SC26, November 15-20, 2026, Chicago, Illinois, USA
979-8-3195-4789-7/26/$31.00 ©2026 IEEE



(a) PageRank on europe_osm (b) GMRES on cage15

Fig. 1: Number of iterations required for algorithm convergence (vertical lines) and kernel performance on H100 (assuming consistent per-iteration kernel execution time). The amortization point in (a) (cuSPARSE–CSR5 crossover) is exact in iteration count but schematic in elapsed time.

which requires more iterations to amortize. Meanwhile, modern numerical algorithms (e.g., advanced preconditioners) are actively reducing required iteration counts [1], widening the gap between kernel design and practical utility.

As ***matrix preprocessing costs rise and iteration counts algorithmically fall***, conventional approaches yields diminishing returns. This paper proposes **SPIKe**, a **SP**arse **I**terative **Ke**rnel generation method that produces practical SpMV kernels in sparse iterative methods, achieving high end-to-end performance across a wide spectrum of iteration counts. SPIKe operates on Compressed Sparse Row (CSR) format and comprises two components: the *Online Policy Template* and the *Offline Code Generator*. Their designs are grounded in the following two key observations.

1) *Sparse matrix preprocessing efforts are essentially equivalent to runtime tile-to-code-path adaptation.* Preprocessing encompasses the pre-computation and storage of matrix sparsity and architecture-specific information. For prevalent CSR-based nnz-split approaches [5], [7], current preprocessing encodes each tile's[1] unique sparse structure into standardized metadata for fitting a single, fixed code path[2]. This pre-standardization regularizes non-zero distribution and enhances runtime load balancing and memory bandwidth efficiency. In this context, costly preprocessing efforts can be equivalently offloaded to online tile-to-code-path adaptation: dynamically selecting the optimal code path for each tile based on its raw inner non-zero pattern. However, with numerous tiles, even simple per-tile operations introduce significant runtime overhead, making lightweight tile pattern analysis essential.

2) *Sparse matrices from the same problem domain exhibit similar sparsity characteristics (domain sparsity), requiring tailored codes.* For nnz-split approaches, we find that tile structure distributions exhibit intra-domain clustering and inter-domain divergence (Fig. 4). When partitioning a sparse matrix into tiles of 512 consecutive non-zeros, nearly all tiles from road domain matrices (e.g., europe_osm, road_usa) span exceeding 128 rows (Fig. 3). In contrast, for biology-domain matrices (e.g., human_gene), this value is almost universally 1. Such tile distribution disparities necessitate different efficient codes for optimal runtime execution. Fortunately, users typically operate within specific domains. This stability enables an offline code-generation stage that is executed once per hardware platform for matrices from the same problem domain, rather than being repeated for every individual matrix.

Guided by these observations, our design is twofold. ***First***, to address the challenge raised at the end of Observation 1, we propose two lightweight *Online Policy Templates*. ***Second***, to exploit the domain sparsity identified in Observation 2, we employ an *Offline Code Generator* to instantiate online templates for specific architecture-domain pairs.

Regarding the online policy template, SPIKe begins with *Thread-Level Parallelism Adaptation* (TPA) that employs lightweight per-tile analysis to select appropriate code paths maximizing load balance and bandwidth utilization. Path selection requires determining cooperative thread values for each tile's row reduction. These values are derived from thread-level row span without data conversions. Additionally, SPIKe designs online *Warp-Level Asymmetric Scheduling* (WAS) policy template to address TPA's imbalance on skewed tiles with super-long rows. WAS introduces *Warp-as-a-Sensor* to perform asymmetry detection solely through register-level shuffle instructions. Upon detecting asymmetry, WAS routes the tile to a dedicated code path with more thread resources for long row reductions.

For the offline code generator, SPIKe employs *Architecture-Domain Co-Aware Code Generation* (ACG) to formalize and instantiate the two aforementioned online policy templates into offline-optimized code. Specifically, ACG performs domain-specific benchmarking and architecture-aware searching to generate specialized high-performance code paths for different domains and platforms. Unlike conventional online preprocessing, which analyzes matrices after program invocation and blocks the application's critical path, our generation occurs entirely before the application runs, eliminating auto-tuning overhead from the online preprocessing.

We evaluate SPIKe at both the kernel and application levels:

1) At the kernel level, SPIKe outperforms non-preprocessing baselines (CUSP [9], LightSpMV [10], Merge-SpMV [11]) with average speedups of 3.97x on H100 and 4.06x on A100. For preprocessing-based baselines (CSR-Adaptive [12], CSR5 [5], cuSPARSE [8]), we evaluate only the isolated kernel GFLOPS, discounting any preprocessing overhead. Although their preprocessing effectively boosts the SpMV kernel performance, SPIKe still maintains a significant edge, achieving 1.90x (H100) and 1.69x (A100) average speedups.

2) At the application level, SPIKe incurs the lowest preprocessing overhead among all evaluated preprocessing-

[1]Tiles refer to matrix partitions created by nnz-split, each containing equal numbers of consecutive non-zeros with different non-zero distributions.

[2]Code path refers to the sequence of instructions executed by a thread block processing its assigned tile; Fixed code path means all thread blocks execute the same instruction sequence regardless of their assigned tile's structure.

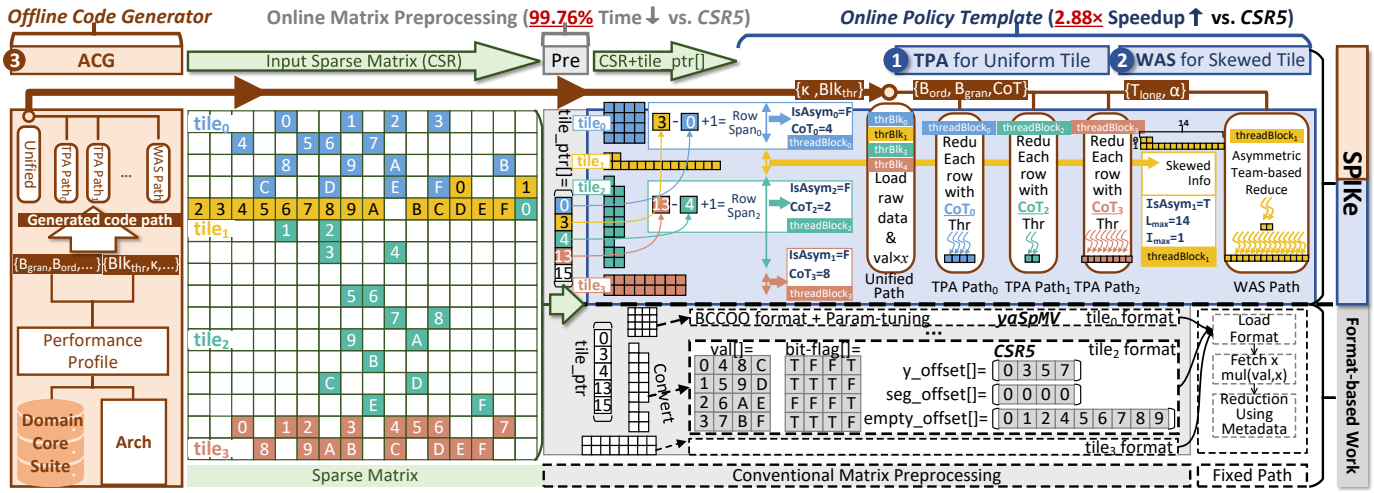


Fig. 2: Overview of SPIKe and comparison with conventional format-based methods (detailed in the bottom-right).

TABLE I: Preprocessing-based CSR5 and preprocessing-free Merge-SpMV performance comparison. The table reports the isolated kernel speedup and the break-even iterations required for CSR5 to achieve a lower end-to-end execution time.

Matrix	CSR5 Kernel Speedup	Break-even N_{iter}
cit-Patents	1.14×	686
mouse_gene	1.35×	252
wiki-Talk	1.21×	913
road_usa	1.11×	155

based implementations. This practical efficiency translates to average end-to-end speedups of 4.26x for GMRES and 2.78x for PageRank over PETSc [13] and SuiteSparse:GraphBLAS [14], respectively. These results comprehensively demonstrate SPIKe's advantages in both kernel performance and practicality. Our contributions include:

- The offline-generation, online-adaptation SPIKe that resolves the contradiction between kernel performance and preprocessing cost in sparse iterative methods.
- Lightweight online policies that perform thread- and warp-level adaptation, dynamically matching each tile to a specialized code path.
- A code generator that offline-produces a library of specialized kernels, leveraging domain sparsity and domain context stability for performance.

II. Background and Motivation

A. CSR and Non-Zero-Based Split

The Compressed Sparse Row (CSR) format represents a sparse matrix via val, col_idx, and row_ptr arrays. Unlike row-splitting, which is sensitive to non-zero distributions, the non-zero-based split (nnz-split) strategy consistently partitions matrices into tiles with equal non-zeros. To enable parallel computation and correct result placement, CSR-based nnz-split methods [5], [7] generate a lightweight tile_ptr array mapping each tile to its starting global row index (Fig. 2).

B. Breaking the Preprocessing Overhead Wall

In sparse matrix computations, preprocessing is a per-matrix online setup that leverages architecture and sparsity information for matrix reordering [3], [15], format conversion [4], [5], and auto-tuning [6], [16]–[18]. For SpMV-based iterative methods, the end-to-end execution time is defined as:

$$T_{e2e} = T_{pre} + N_{iter} \times T_{spmv} + T_{other} \quad (1)$$

where T_{pre} is the preprocessing overhead, N_{iter} is the iteration count, T_{spmv} is the single-kernel execution time, and T_{other} represents the remaining algorithmic costs. As shown in Table I, while preprocessing-based CSR5 [5] achieves a 1.11x–1.35x kernel speedup over preprocessing-free Merge-SpMV [11], it requires 155–913 iterations to break even.

For applications requiring few N_{iter}—such as iterative solvers with advanced preconditioners, well-conditioned linear systems, or approximate PageRank—the preprocessing overhead easily dominates T_{e2e}, nullifying kernel-level performance gains. We refer to this phenomenon as the ***preprocessing overhead wall***. This wall fundamentally limits the practicality of high-performance sparse operators. Breaking through it requires rethinking what conventional preprocessing accomplishes and how online equivalent optimizations can be executed, which motivates SPIKe.

III. Design Overview of SPIKe

The nnz-split strategy partitions sparse matrices into tiles with equal non-zeros, yet these tiles (e.g., $\text{tile}_0 - \text{tile}_3$ in Fig. 2) exhibit different non-zero distributions. To manage this, costly conventional GPU-based preprocessing [4], [5], [7] converts each unique tile into a standardized format via complex metadata augmentation (e.g., tile_2 metadata in CSR5 [5], bottom-right of Fig. 2). While this sophisticated standardization optimizes load balancing and memory access, it forces execution through a fixed code path (*Fixed Path*, Fig. 2). Methods like yaSpMV [4] employ more complex online auto-tuning to determine parameters during preprocessing.

Algorithm 1: SPIKe Overview

```
// Offline Stage
// ACG(Arch, Domain) ⇒ {Blk_thr, κ, B_gran, B_ord, α, T_long}
Input  : x[], CSR{row_ptr[], val[], col_idx[]}
Output: y[]
1  __shared__ s_mid[Tile_nnz]; // Tile_nnz ← Blk_thr × κ
2  tile_start_nnz ← Tile_nnz × blockIdx.x;
   /* Unified Loading & Multiplication Block                    */
3  for i ← 0 to κ do
4  |   lid ← i × Blk_thr + threadIdx.x;
5  |   gid ← tile_start_nnz + lid;
6  |   s_mid[lid] ← val[gid] × x[col_idx[gid]];
7  __syncthreads();
8  if tile is uniform then
9  |   ⟨T_TPA^{B_gran, B_ord}⟩(row_ptr, s_mid, y) // TPA Policy Alg.2
10 if tile is skewed then
11 |   ⟨T_WAS^{α, T_long}⟩(row_ptr, s_mid, y) // WAS Policy Alg.3
```

In contrast, SPIKe dynamically adapts code paths to raw CSR structures, eliminating heavy format conversions and online auto-tuning. As shown in Fig. 2, the online preprocessing of CSR5 includes the generation of tile_ptr, the generation of tile_desc, and the transposition of the col_idx and val arrays. On GPU platforms, the latter two steps account for the larger part of this cost [5]. In contrast, the online preprocessing of SPIKe includes only the lightweight tile_ptr construction, which merely stores each tile's starting row index. It does not build complex tile descriptors, perform permutation, or conduct online parameter tuning. SPIKe compensates for the lack of complex metadata through two key components: an *Online Policy Template* that shifts load balancing and memory access optimization to runtime tile-to-code-path adaptation, and an *Offline Code Generator* that moves matrix-aware auto-tuning entirely offline.

Online Policy Template matches structurally distinct tiles to specialized code paths at runtime (Fig. 2). Initially, all thread blocks execute a unified, bandwidth-centric code path for element-wise data loading and multiplication (Section IV-A). For the subsequent row-wise reduction, SPIKe performs a lightweight tile-structure analysis to dynamically assign each tile a specialized code path with appropriate thread allocation. Uniform tiles trigger *Thread-Level Parallelism Adaptation* (TPA; Section IV), which defines a `row_span`-based template to dynamically determine the cooperative thread size. In contrast, highly skewed tiles activate *Warp-Level Asymmetric Scheduling* (WAS; Section V), applying an asymmetric processing template based on the *Warp-as-a-Sensor* mechanism.

For the *Offline Code Generator*, our *Architecture-Domain Co-Aware Code Generation* (ACG; Section VI) formalizes these TPA and WAS templates into a parameterizable solution space. It then performs a profile-guided search to find the optimal parameters and instantiates the final kernel for the target architecture-domain pair. As summarized in Fig. 2 and Algorithm 1, TPA ❶ (line 9) and WAS ❷ (line 11) govern the online tile-to-code-path routing. ACG ❸ leverages architecture and domain information offline to instantiate these online policy templates and generate optimized code.

IV. Thread-Level Parallelism Adaptation

Online Policy Template in this section and the next defines how to adapt to tile non-zero patterns at runtime. The Thread-Level Parallelism Adaptation (TPA) technique dynamically assigns each uniform tile to the tailored cooperating threads code path based on its lightweight characteristics.

A. Unified Loading and Multiplication Code Path

The SpMV computation comprises two logical stages: 1) element-wise data loading and multiplication, and 2) row-wise reduction. Because the first stage lacks non-zero dependencies, its inherent parallelism is completely independent of the tile structure. Consequently, SPIKe implements a unified, bandwidth-centric code path well-suited for this stage across all tiles (Algorithm 1, lines 3-6). Mapping one thread block to each tile ensures adjacent threads access consecutive global indices (gid), maximizing memory bandwidth utilization for the val and col_idx arrays (Line 6). Specifically, we set the number of non-zeros per tile ($Tile_{\mathsf{nnz}}$) to $\kappa \times Blk_{\mathsf{thr}}$, where the multiplier κ and thread block size Blk_{thr} are offline-tuned parameters. Table II summarizes them along with other parameters, which will be detailed later.

B. Tile-Aware Divergence-Free Reduction Code Path

TPA dynamically selects the appropriate Cooperative Thread group (`CoT`) size for each tile's row-reduction based on its `row_span`. As shown in Algorithm 2, this runtime logic navigates a multi-branch code structure. Because each thread block exclusively processes a single tile, all threads within the block evaluate an identical `row_span`, ensuring inherently divergence-free execution. This multi-branch structure is configured offline by the code generator through branch granularity (B_{gran}), branch order (B_{ord}) and `CoT`.

1) Thread-Level Tile Row Span: Unlike the data-loading stage, row-wise reduction involves data dependencies, making the optimal reduction code path dependent on the tile structure. While existing methods like CSR-Vector [19], [20] assign `CoT` based on the average row length, computing this per tile requires costly integer division operations. Instead, we use the tile's row span (the number of rows it covers)—efficiently computed as $\mathsf{tile\_ptr}[i{+}1]{-}\mathsf{tile\_ptr}[i]{+}1$—as a runtime proxy. Because each tile contains a fixed number of consecutive non-zeros, row span exhibits a strong negative correlation with average row length: a small span indicates a *short-and-fat* tile structure, whereas a large span indicates a *tall-and-skinny* one. Instead of utilizing hardware intrinsics (e.g., `__clz`), we adopt a multi-branch structure configured by B_{gran} and B_{ord}. This design enables static compiler optimization by leveraging compile-time `CoT` for aggressive loop unrolling, refined register allocation, and hard-coded warp-shuffle offsets.

2) Branch Granularity Search Space: B_{gran} dictates how finely the `row_span` ranges are partitioned into distinct `if` branches. A finest-grained strategy (Algorithm 2, Lines 2-6) partitions tiles into as many valid ranges as possible to maximize specialization, which explicitly balances computational loads and benefits GPUs with limited warp scheduling

TABLE II: SPIKe's key tunable parameters.

Category	Description	Symbol	Section
TPA	Branch granularity	B_{gran}	§ IV-B2
	Branch ordering	B_{ord}	§ IV-B3
	Cooperative threads	CoT	§ IV-B1
WAS	Skewness factor	α	§ V-A
	Long-Row team size	T_{long}	§ V-B
Arch param	Parallel granularity	Blk_{thr}	§ IV-A
	Memory access granularity	κ	

Algorithm 2: TPA policy template $\mathcal{T}_{\text{TPA}}$ instance

Input : row_ptr[], tile_ptr[], s_mid[]
Output: y[]
// **Offline Config:**
ACG(Arch, Domain) $\Rightarrow Blk_{\text{thr}} = 128, \kappa = 4, B_{\text{gran}}, B_{\text{ord}}$
// **Online Execution:** Launched Blk_{thr} threads per block
// Alg.1 Unified Loading & Multiplication Block
1 `rs` $\leftarrow$ tile_ptr$[blockIdx.x+1]$ − tile_ptr$[blockIdx.x]$; // row_span
// Generated REDUCE_BLOCK: if-else chain ordered by Fig. 3
2 **if** `rs` $\in [64, 128)$ ***[[likely]]*** **then**
`reduceRowCodePath`⟨CoT = 2⟩(row_ptr, s_mid, y) ;
3 **else if** `rs` $\in [8, 16)$ **then** `reduceRowCodePath`⟨CoT = 16⟩(...) ;
4 **else if** `rs` $\in [16, 32)$ **then** `reduceRowCodePath`⟨CoT = 8⟩(...) ;
5 **else if** `rs` ≥ 128 **then** `reduceRowCodePath`⟨CoT = 1⟩(...) ;
// ... other customized branches omitted for brevity ...
6 **else if** `rs` $= 1$ **then** `reduceRowCodePath`⟨CoT = 128⟩(...) ;

efficiency. Conversely, higher-performance architectures favor coarser granularity (i.e., merging ranges). The CoT is bounded by lower and upper limits. The upper bound prioritizes sufficient intra-row workload parallelism to maximize intra-row reduction efficiency:

$$\texttt{CoT}_{i,max} = 2^{\lfloor \log_2(Tile_{\text{nnz}}/\texttt{row\_span}(i)) \rfloor} \tag{2}$$

The lower bound prioritizes thread resource utilization by maximizing inter-row parallelism:

$$\texttt{CoT}_{i,min} = 2^{\lceil \log_2(Blk_{\text{thr}}/\texttt{row\_span}(i)) \rceil} \tag{3}$$

Algorithm 2 utilizes this lower bound. Any $\texttt{CoT}_i$ within the range $[\texttt{CoT}_{i,min}, \texttt{CoT}_{i,max}]$ represents a viable configuration for offline code generation search.

3) Statistical Branch Order Optimization: While B_{gran} determines the number of condition branches, B_{ord} dictates their evaluation sequence. Leveraging the aggregate row_span distribution across commonly used datasets in Table III, SPIKe generates an explicit if-else chain (Algorithm 2) ordered by descending tile counts in Fig. 3. As illustrated in Algorithm 2, evaluating the statistically dominant range ($[64, 128)$) first enables early exit for the majority of thread blocks, thereby minimizing average condition checks. Furthermore, annotating these branches with *[[likely]]* attributes guides the compiler to prioritize these hot paths in the physical machine-code layout. However, universal optimization for datasets across diverse domains is sub-optimal. As shown in Fig. 4, row_span distributions cluster tightly within the same domain but diverge across domains. In practice, SPIKe replaces lines 2-6 of Algorithm 2 with domain-specialized code paths in Fig. 4, enabling structural variations to drive tailored kernel implementations.

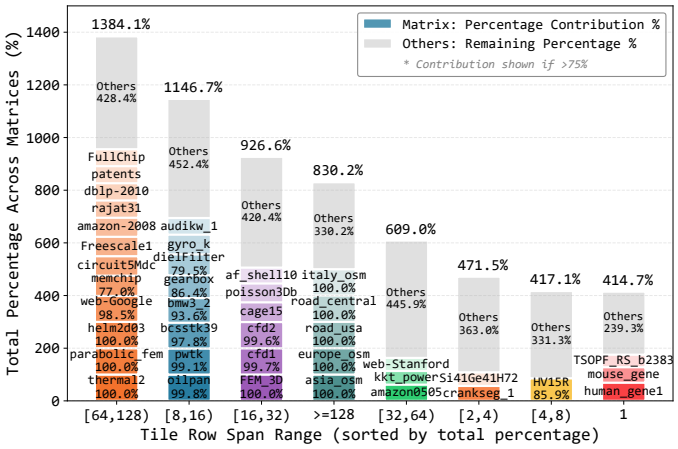


Fig. 3: Row span range percentage from Table III set.

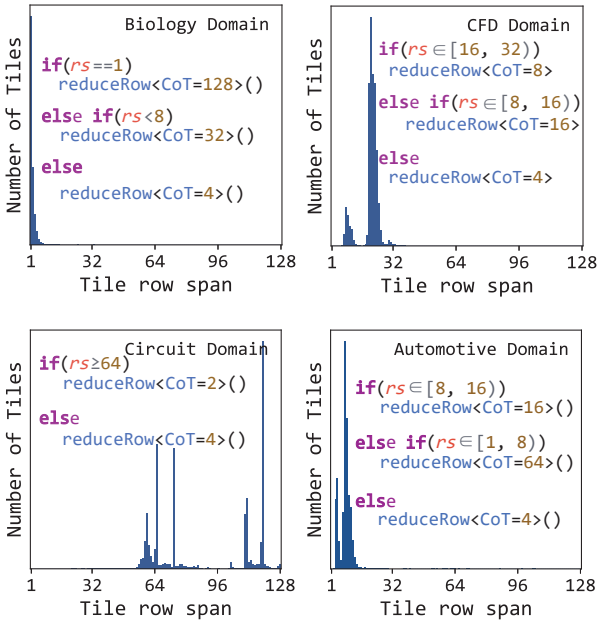


Fig. 4: Tile row span distributions (histograms) and corresponding domain-specialized generated REDUCE_BLOCK code for four domains in Table III ($Tile_{\text{nnz}} = 512, Blk_{\text{thr}} = 128$), where rs denotes row span, and reduceRow abbreviates reduceRowCodePath. Vertical axes omit specific values to highlight relative proportions; row spans exceeding Blk_{thr} share a uniform branch, truncating the horizontal axis.

Collectively, these parameters define a vast optimization space: B_{gran} modulates branch merging, B_{ord} adapts to domain statistics, and CoT spans a bounded viable range. SPIKe navigates this space via the offline code generation described in Section VI to synthesize optimized kernels.

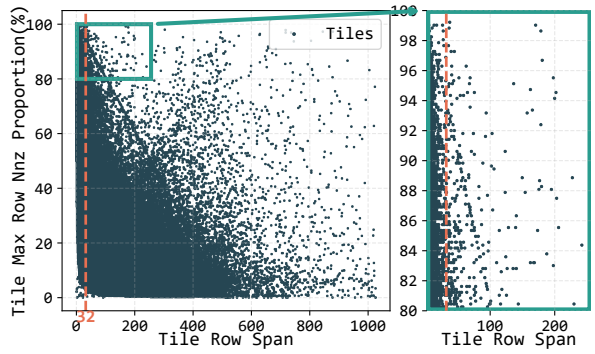


Fig. 5: Ratio of max row length to $Tile_{\text{nnz}}$ for each tile.

V. Warp-Level Asymmetric Scheduling

While TPA handles uniform tiles using fixed CoT threads per row, highly skewed tiles (e.g., tile_1 in Fig. 2) induce severe load imbalance. Uniform thread distribution causes threads on short rows to idle prematurely, leaving threads on long rows as performance bottlenecks. To mitigate this underutilization, SPIKe introduces Warp-Level Asymmetric Scheduling (WAS).

A. Warp-as-a-Sensor Asymmetry Detection

Runtime analysis of row length distribution within each tile incurs significant overhead. Determining the number of non-zeros per row requires $2 \times$ row_span global memory accesses to the row_ptr, while subsequent comparisons and decisions need expensive thread-block synchronization.

Analyzing matrices from Table III ($Tile_{\text{nnz}} = 512$), we define skewed tiles as those in which the longest row contains over 80% of the non-zeros. As shown in Fig. 5, 99.91% of these highly skewed instances have a row span ≤ 32, and skew probability drops as row span increases. Consequently, ***severe load imbalance almost exclusively occurs in tiles spanning no more than 32 rows***. This motivates our *Warp-as-a-Sensor* design, which exploits the 32-thread parallelism of one GPU warp to efficiently and accurately detect skewed tiles.

For tiles with row_span ≤ 32, the first warp performs this detection (Algorithm 3, Line 2). Each thread calculates the number of non-zeros for its assigned row (Line 3). The warp aggregates the maximum row length (L_{max}) within the tile, its local index (I_{max}), and the boolean IsAsym (Lines 4-6) with shuffle instructions, bypassing shared memory and block synchronization. A tile is classified as skewed if $\text{L}_{\text{max}} > \alpha \cdot Tile_{\text{nnz}}$, where α is a tunable skewness threshold.

B. Asymmetric Team-Based Reduction Code Path

Tiles detected as skewed route to a specialized reduction code path that asymmetrically divides the thread block into two collaborative teams based on row length.

1) Long-Row Team: This team consists of the majority of threads, with the number of threads T_{long} determined via profile-guided search. We use the following Equation to find an initial estimate.

Algorithm 3: $\mathcal{T}_{\text{WAS}}$ template (tile row_span ≤ 32)

Input : row_ptr[], $Tile_{\text{nnz}}$, tile_ptr[], s_mid[]
Output: y[]

1 __*shared*__ s_info[3], $s\_final[32]$, $s\_warp\_sums[Blk_{\text{thr}}/32]$;
// Phase 1: Warp-as-a-Sensor Skewness Detection
2 **if** $(threadIdx.x \gg 5) == 0$ **then**
3 $len \leftarrow$ GetLocalRowLength($Tile_{\text{nnz}}$, tile_ptr, row_ptr);
// Register-level shuffle to find max row length and its index
4 $(\text{L}_{\text{max}}, \text{I}_{\text{max}}) \leftarrow$ WarpReduceMax($len, threadIdx.x$);
5 **if** $threadIdx.x == 0$ **then**
6 s_info $\leftarrow$ {IsAsym=($\text{L}_{\text{max}} > \alpha \cdot Tile_{\text{nnz}}$), L_{max}, I_{max}};
7 __syncthreads();
// Phase 2: Asymmetric Team-Based Reduction
8 **if** *s_info*.IsAsym $= true$ **then**
9 **if** $threadIdx.x < T_{long}$ **then**
// Long-Row Team: Collaborative reduction on I_{max}
10 $thr\_sum \leftarrow$ StridedSum(s_info.I_{max}, T_{long}, s_mid);
11 $warp\_sum \leftarrow$ WarpReduceSum(thr_sum);
12 **if** $threadIdx.x \ \& \ 31 == 0$ **then**
13 $s\_warp\_sums[threadIdx.x \gg 5] \leftarrow warp\_sum$;
14 **else**
// Short-Row Team: Independent polling on other rows
15 $team\_tid \leftarrow threadIdx.x - T_{long}$;
16 **for** $i \leftarrow team\_tid$ **to** row_span ***step*** $(Blk_{\text{thr}} - T_{long})$ **do**
17 **if** $i \neq \text{I}_{\text{max}}$ **then**
18 $s\_final[i] \leftarrow$ ProcessShortRow(i, s_mid);
19 __syncthreads();
// Aggregate Long-Row warp sums and write back
20 **if** $threadIdx.x < 32$ **then**
21 $nwarps \leftarrow T_{long} \gg 5$; $val \leftarrow (threadIdx.x < nwarps)$? $s\_warp\_sums[threadIdx.x]$: 0;
22 $final\_sum \leftarrow$ WarpReduceSum(val);
23 **if** $threadIdx.x == 0$ **then**
24 $s\_final[\text{s\_info.I}_{\text{max}}] \leftarrow final\_sum$;
25 __syncthreads();
26 **if** $threadIdx.x <$ row_span **then**
27 $y[\text{tile\_ptr}[blockIdx.x] + threadIdx.x] \leftarrow s\_final[threadIdx.x]$;
28 **else**
29 reduceRowCodePath⟨CoT⟩(row_ptr, s_mid, y); // Alg. 2

$$\hat{T}_{long} \propto \left(\left(\frac{L_{max}}{Tile_{\text{nnz}}} \right) \cdot Blk_{\text{thr}} \right) \tag{4}$$

where the final $\hat{T}_{long}$ is constrained to a warp size multiple for hardware efficiency. In practice, we conduct a local search within a constrained window around this estimate ($\hat{T}_{long} \pm 2$ warps) to determine T_{long}, enabling resource allocation to match the actual skewness and underlying hardware specifics.

2) Short-Row Team: This team consists of the remaining threads and processes all other short rows within the tile (Lines 15-19). Each thread independently processes one or more short rows serially in a polling manner. Upon completion, both teams synchronize at the block level to aggregate results in shared memory (Lines 20-25) before writing to the output vector (Lines 26-27).

VI. Architecture-Domain Co-Aware Code Generation

Our code generation (ACG) involves two stages: *Policy Formalization* (Fig. 6(1)) and *Profile-Guided Instantiation* (Fig. 6(2)). Formalization translates the abstract online TPA and WAS policies into a concrete, parameterizable solution

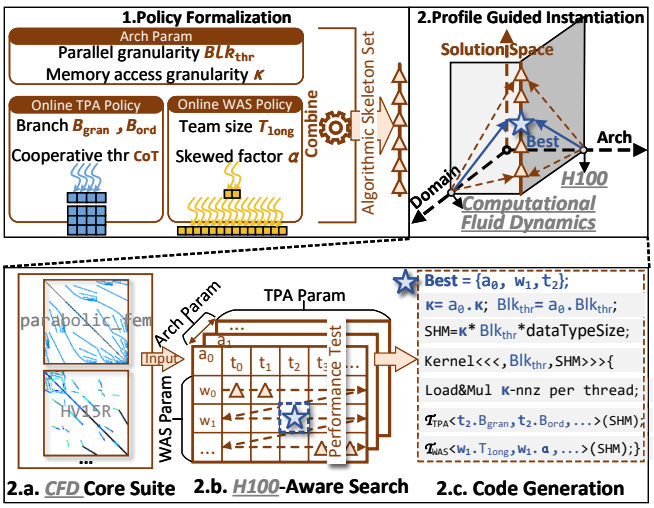


Fig. 6: Architecture-Domain Co-Aware Code Generation for the Computational Fluid Dynamics (CFD) domain on H100.

space called the *Algorithmic Skeleton Set*. Profile-Guided Instantiation then systematically searches this space to generate an optimized kernel for a specific architecture-domain pair. This generation is an ahead-of-time process that occurs only once per architecture and domain. Unlike runtime matrix preprocessing, it tunes a representative batch of matrices rather than individual ones. Users can run the ACG asynchronously at any time before deployment to synthesize a standalone, domain-specialized SpMV library. At runtime, iterative methods simply link this pre-compiled library, achieving plug-and-play execution with zero online tuning or formatting overhead.

A. Domain Sparsity and Domain Context Stability

Sparse matrices from different applications exhibit distinct structural characteristics [21]. Linear solvers typically process physical simulation matrices with favorable locality, whereas graph analytics often handle irregular, power-law network topologies [22]. As shown in Fig. 3, road domain matrices exhibit tile row spans ≥ 128, while biology matrices concentrate around a span of 1. These domain-specific characteristics, termed *domain sparsity*, directly inform our code generation. Fortunately, users generally operate within stable domain contexts (e.g., a biologist rarely processes both gene matrices and social network graphs within the same workflow). This stability enables domain-specific offline optimization with tailored parameters, an opportunity overlooked by existing auto-tuning frameworks [6], [16]–[18], [23]. SPIKe also provides a default configuration to facilitate immediate use, offering robust performance without any offline search.

B. Policy Formalization

Policy Formalization identifies structural parameters within the abstract TPA and WAS policies to define a finite, parameterizable solution space: the *Algorithmic Skeleton Set* (Fig. 6(1)). Within this space, B_{gran}, B_{ord}, and CoT govern the TPA multi-branch reduction structure. We pre-define five B_{gran} strategies ranging from coarse- to finest-grained levels, while B_{ord} arranges the branch order based on domain statistics, and CoT dictates intra-row parallelism. For handling skewed tiles via WAS, α and T_{long} control the skewness detection and asymmetric resource allocation, respectively. Thread block size ($Blk_{\text{thr}} \in \{128, 256, 512\}$) impacts hardware occupancy and resource allocation. Memory access granularity ($\kappa \in \{2, 4, 8\}$) sets the work per thread, governing memory bandwidth efficiency and shared memory utilization. These parameters transform the abstract policies into a concrete solution space navigated during instantiation.

C. Profile-Guided Instantiation

Finding the optimal kernel configuration for each architecture-domain pair requires systematically exploring the *Algorithmic Skeleton Set*. As shown in Fig. 6(2), we employ a profile-guided approach driven by Simulated Annealing (SA) to navigate this space efficiently. This process is as follows:

1) Domain-Specific Benchmarking: To capture structural diversity and scale variance within a domain, we construct a domain core suite (Fig. 2 ❸, Fig. 6(2.a)) via K-means clustering in a 2D feature space (log-transformed row dimension and average non-zeros per row). By selecting the matrix nearest to each of the K centroids, and dynamically scaling K with the domain size to balance search overhead and representation accuracy, we form a representative suite. This suite evaluates candidate configurations, providing performance feedback to reliably capture the target domain sparsity.

2) Architecture-Aware SA Search: Our SA-based search iteratively explores the solution space using the domain core suite directly on the target hardware (Fig. 6(2.b)). This architecture-aware profiling incurs a strict one-time offline cost per architecture-domain pair. For example, profiling a 5-matrix core suite for the *Computational Fluid Dynamics* domain takes $\sim$2 minutes on an H100 GPU. Given the 174 matrices within this domain in the SuiteSparse Collection [24], this purely offline overhead is justified and entirely decoupled from runtime execution.

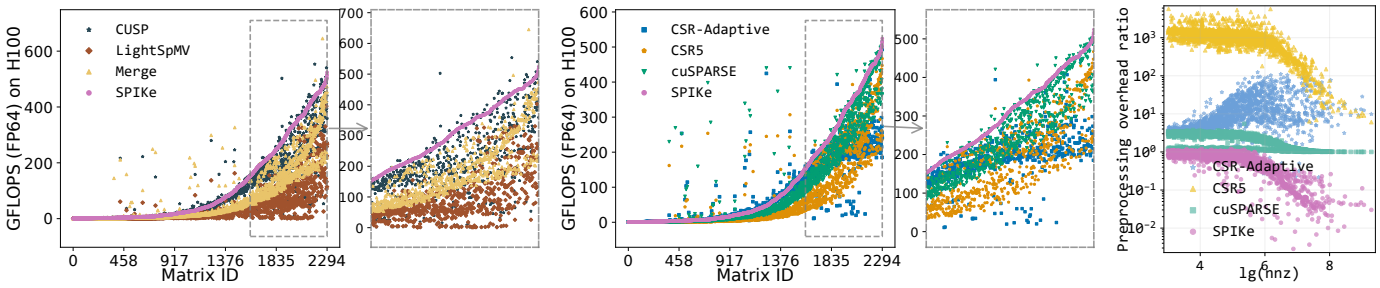
3) Metaprogramming-based Code Generation: Once SA identifies the well-suited configuration, SPIKe generates the final code using the *Jinja2* templating engine [25]. This metaprogramming approach directly embeds the optimal parameters (e.g., κ, Blk_{thr}, and selected reduction branches) into the kernel source code as constants (Fig. 6(2.c)). Consequently, the compiler leverages constant propagation to produce highly efficient instruction sequences and reduce register pressure. Similar techniques have proven effective in high-performance libraries like ATLAS [26] and FFTW [27].

VII. Evaluation

A. Setup

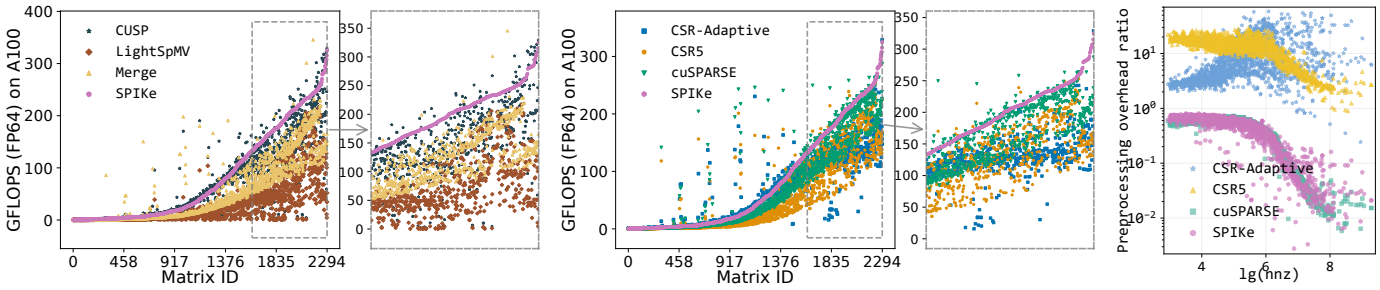
Platform. We conduct experiments on two NVIDIA GPUs: H100 (Hopper) 80GB HBM3 and A100 (Ampere) 80GB PCIe. All codes are compiled using NVCC from CUDA 12.8.

Baselines. We compare SPIKe against six state-of-the-art SpMV implementations, divided into two categories



(a) Baselines without preprocessing (b) Baselines with preprocessing (c) Preprocess Overhead

Fig. 7: Comparison of kernel performance and preprocessing overhead ratio on H100 (matrices are sorted by SPIKe GFLOPS).



(a) Baselines without preprocessing (b) Baselines with preprocessing (c) Preprocess Overhead

Fig. 8: Comparison of kernel performance and preprocessing overhead ratio on A100 (matrices are sorted by SPIKe GFLOPS).

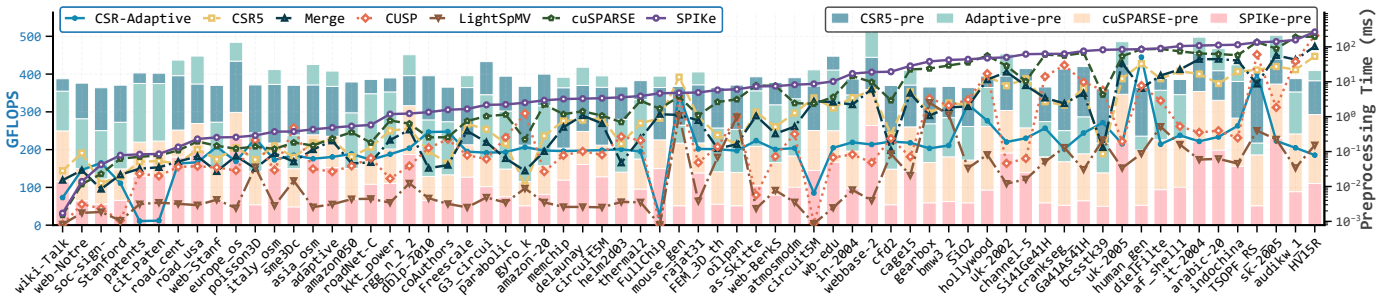


Fig. 9: SpMV kernel performance (lines, GFLOPS) and preprocessing time (bars, ms) in FP64 on Table III test set for H100.

based on preprocessing requirements: without preprocessing—1) CUSP [9] 2) LightSpMV [10], and 3) Merge-SpMV (Merge) [11]; with preprocessing—4) CSR-Adaptive [12], 5) CSR5 [5] and 6) cuSPARSE 12.8 [8]. For a fair comparison, we call *cusparseSpMV_preprocess* to accelerate subsequent cuSPARSE SpMV operations, which aligns with our target scenario of multiple SpMV calls on matrices with static sparsity patterns.

Dataset. We evaluate SpMV kernel performance using 2,294 sparse matrices from the SuiteSparse Matrix Collection [24]. For detailed performance comparison and application evaluation, we select representative datasets commonly used in previous SpMV work [21], [28], as shown in Table III.

B. Isolated Kernel Performance and Preprocessing Overhead

Fig. 7(a) and (b) show the SpMV kernel performance comparison in FP64 on H100, with matrices sorted by SPIKe GFLOPS. SPIKe demonstrates significant average speedups over baselines without preprocessing: 3.87x versus CUSP, 5.74x versus LightSpMV, and 2.31x versus Merge-SpMV. Against baselines with preprocessing, SPIKe achieves average speedups of 1.24x over CSR-Adaptive, 2.88x over CSR5, and 1.58x over cuSPARSE. Fig. 8(a) and (b) present results on A100. SPIKe achieves high performance with average speedups of 2.68x over CUSP, 7.46x over LightSpMV, 2.06x over Merge-SpMV, 1.21x over CSR-Adaptive, 2.41x over CSR5, and 1.46x over cuSPARSE. The results reveal that preprocessing-based baselines generally outperform those without preprocessing. This advantage arises from the benefits that preprocessing provides to the SpMV kernel.

Online preprocessing is the per-matrix runtime setup that is performed after the sparse matrix data becomes available and before one or more SpMV executions. It inspects the matrix and produces reusable performance artifacts, such as reordered indices, converted formats, and auxiliary metadata. While achieving the highest average SpMV kernel perfor-

TABLE III: Commonly used representative test set.

Graph Analytics				Scientific Computing			
Domain	**Matrix**	**Dim**	**nnz**	**Domain**	**Matrix**	**Dim**	**nnz**
Road	europe_osm	51M	108M	CFD	atmosmodm	1M	10M
	road_central	14M	34M		cfd2	123K	3M
	road_usa	24M	58M		HV15R	2M	283M
	roadNet-CA	2M	6M		poisson3Db	86K	2M
	italy_osm	7M	14M		parabolic_fem	526K	4M
	asia_osm	12M	25M	Circuit	circuit5M	6M	60M
Web	web-Google	916K	5M		memchip	3M	15M
	web-Stanford	282K	2M		G3_circuit	2M	8M
	webbase-2001	118M	1G		FullChip	3M	27M
	uk-2002	19M	298M		Freescale1	3M	19M
	indochina-2004	7M	194M		circuit5M_dc	4M	19M
	arabic-2005	23M	640M		rajat31	5M	20M
	in-2004	1M	17M	EM	dielFilterV2real	1M	49M
	web-BerkStan	685K	8M	DIM	channel-500	5M	85M
	Stanford	282K	2M		adaptive	7M	27M
	sk-2005	51M	2G	Physics	helm2d03	392K	3M
	it-2004	41M	1G	Mech	crankseg_1	53K	11M
	uk-2005	39M	936M		gearbox	154K	9M
	wb-edu	10M	57M		gyro_k	17K	1M
	web-NotreDame	326K	1M		sme3Dc	43K	3M
Social	hollywood-2009	1M	114M		af_shell10	2M	53M
	soc-sign-epinions	132K	841K	Struct	bcsstk39	47K	2M
	wiki-Talk	2M	5M		pwtk	218K	12M
	amazon-2008	735K	5M	Auto	bmw3_2	227K	11M
	amazon0505	410K	3M		oilpan	74K	4M
Citation	coAuthorsDBLP	299K	2M		audikw_1	944K	78M
	cit-Patents	4M	17M	Chem	Ga41As41H72	268K	18M
	patents	4M	15M		Si41Ge41H72	186K	15M
	dblp-2010	326K	2M		SiO2	155K	11M
Graph	rgg_n_2_24_s0	17M	265M	Thermal	thermal2	1M	9M
Geom	delaunay_n23	8M	50M		FEM_3D_thermal2	148K	3M
Network	as-Skitter	1M	22M	Biology	mouse_gene	45K	29M
					human_gene1	22K	25M
					cage15	5M	99M
				Power	TSOPF_RS_b2383	38K	16M
					kkt_power	2M	15M

Abbreviations: Geom = Geometry, CFD = Computational Fluid Dynamics, EM = Electromagnetic, DIM = DIMACS Challenge, Mech = Mechanical, Struc = Structural, Auto = Automotive, Chem = Chemistry.

mance, SPIKe also maintains nearly negligible preprocessing overhead. It is worth noting that SPIKe's online preprocessing only includes the lightweight tile_ptr construction (Section III). Offline ACG is not online preprocessing. It is a one-time, ahead-of-time code generation step for each architecture-domain pair, and it lies outside the critical path of application deployment. TPA and WAS are in-kernel code-path selections during SpMV, and their cost is included in the measured SpMV kernel time. Fig. 7(c) and Fig. 8(c) illustrate the preprocessing overhead ratio, defined as preprocessing time divided by single SpMV execution time. cuSPARSE shows stable and lightweight preprocessing overhead across different matrix sizes on H100. CSR5 performs better for large-scale matrices, while CSR-Adaptive is more efficient for small-scale matrices. Figs. 9 and 10 provide a comprehensive side-by-side comparison of kernel performance and actual preprocessing time (on a logarithmic scale) between SPIKe and six baselines on the representative test set for both platforms. SPIKe achieves the highest GFLOPS performance and lowest preprocessing overhead across most test cases; for instance, on the H100, its preprocessing time is reduced by 99.76% compared to CSR5.

Fig. 11 reports the kernel performance and preprocessing time of SPIKe and of two Tensor-Core-based SpMV baselines on H100. SPIKe achieves an average speedup of 1.32x over DASP [29] and of 1.24x over Drawloom [30]. Both baselines incur substantial preprocessing overhead, larger than that of CSR5. On average, it is more than four orders of magnitude longer than the preprocessing time of SPIKe. This long preprocessing time in DASP and Drawloom arises from reorganizing the CSR format into a Tensor-Core-friendly layout—grouping rows by nonzero count, packing rows into MMA tiles, and rearranging nonzeros into the fragment layout—which requires traversing nonzeros several times and materializing a padded structure larger than the original CSR.

C. Real Application End-to-End Performance

1) PageRank Algorithm: We implement a PageRank algorithm following the LAGraph [31] standard (damping=0.85, tolerance=1e-4) and integrate SPIKe into this implementation. For fair comparison, we use PLUS_TIMES semiring instead of plus-second when comparing against SuiteSparse: GraphBLAS [14]. Fig. 12 shows that across *graph analytics* matrices from Table III with iteration counts ranging from 5 to 61, the SPIKe-based implementation achieves a 2.78x average speedup over GraphBLAS.

2) GMRES Method: SPIKe is integrated into PETSc by encapsulating its kernel as a MatShell operator. We run GMRES with Jacobi preconditioning using restart r=30, relative tolerance 1e-6, and maximum iterations 2000. We compare PETSc's AIJ against SPIKe-backed MatShell on the same matrices. The evaluation metric is total wall-clock solve time. Fig. 13 compares the implementation integrating SPIKe SpMV against PETSc using 6 convergent sparse matrices from the *scientific computing* of Table III. On H100, SPIKe achieves speedups ranging from 1.26x to 17.43x with a 4.26x average acceleration. SPIKe delivers performance gains across different iteration counts, from fast-converging (11-53 iterations) to slow-converging cases (162-1635 iterations). With SPIKe, the total solve time reduces from 40.17 seconds to 9.4 seconds while maintaining identical convergence behavior. These results demonstrate that SPIKe effectively accelerates iterative solvers across diverse iteration counts.

D. Ablation Study

In this section, we investigate how SPIKe benefits from different optimizations.

1) SPIKe Performance Breakdown: We illustrate the SPIKe performance breakdown on H100 in Fig. 14. To isolate the benefits of online policies, the *nnz-split* only retains the equal-nnz macro-partitioning but disables TPA and WAS, reverting to a static intra-tile mapping where each thread independently reduces one row; *online only* employs heuristic-based default configurations to instantiate the TPA (with the general Algorithm 2 ordering) and WAS ($\alpha = 0.8$) policies. Across the evaluated matrices, the *Online Policy Template* (TPA+WAS) and *Offline Code Generator* contribute 27% and 18% of SPIKe's final performance. To further explore the detailed acceleration mechanisms, we conduct ablation studies as shown in Figs. 15, 16.

2) Warp-Level Asymmetric Scheduling Effectiveness: Fig. 15 arranges sparse matrices by average maximum row fraction. This fraction represents the ratio of non-zero counts

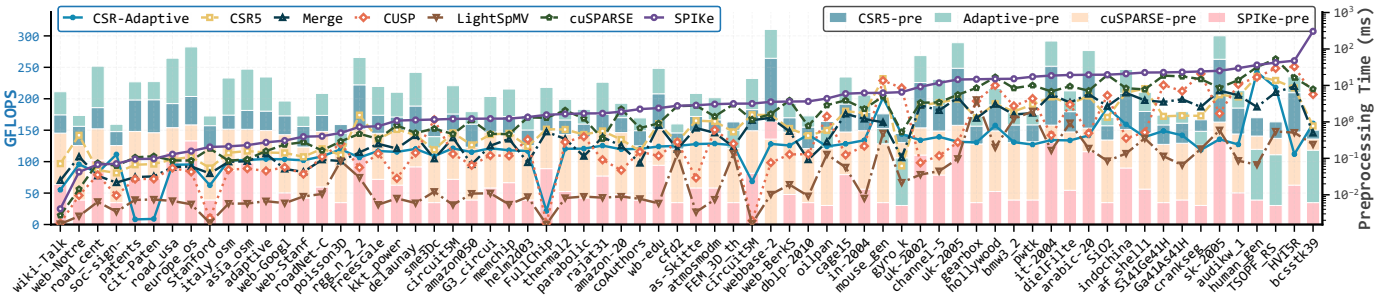


Fig. 10: SpMV kernel performance (lines, GFLOPS) and preprocessing time (bars, ms) in FP64 on Table III test set for A100.

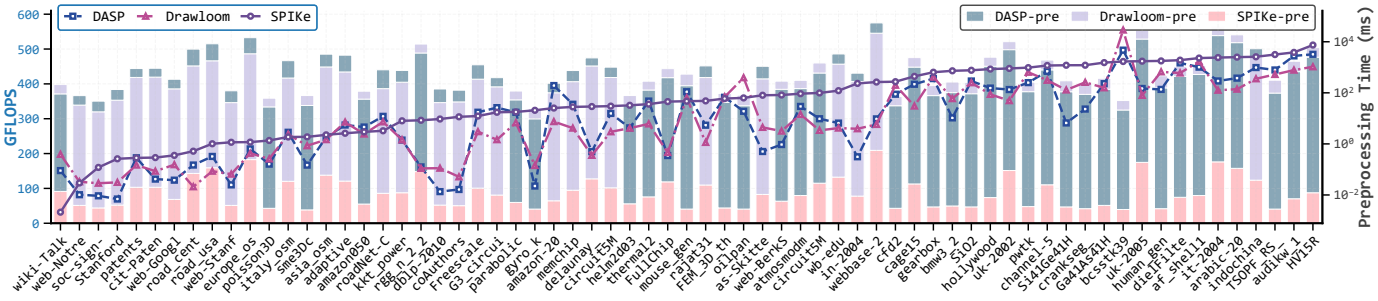


Fig. 11: SpMV kernel performance (lines, GFLOPS) and preprocessing time (bars, ms) in FP64 on H100, comparing SPIKe against Tensor-Core-based baselines.

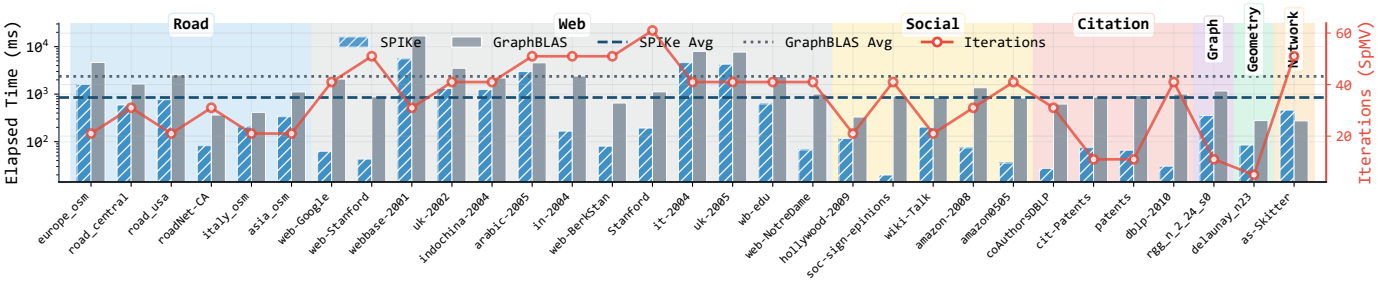


Fig. 12: PageRank performance (FP32) comparison on H100.

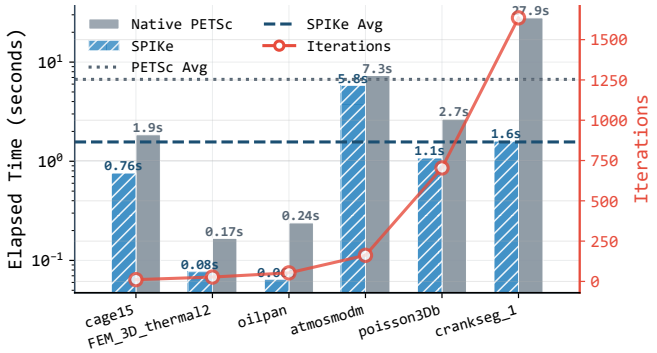


Fig. 13: GMRES performance (FP64) comparison on H100.

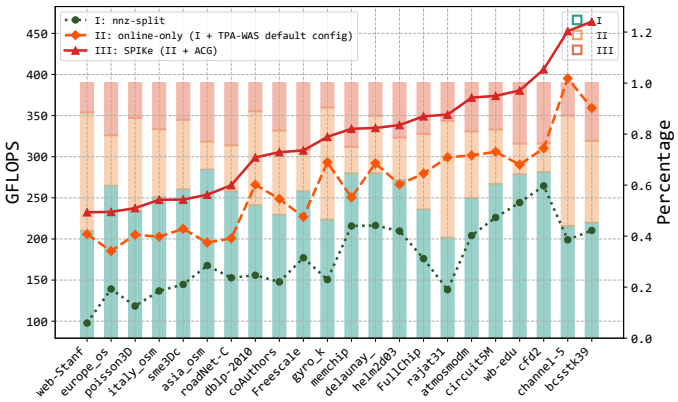


Fig. 14: Performance breakdown of SPIKe.

in the longest row to the total non-zeros in each tile. Higher fractions indicate more skewed tiles (human_gene). WAS significantly improves performance for matrices with numerous skewed tiles. For matrices like europe_osm with primarily uniform tiles, WAS shows no performance improvement since it targets skewed tile optimization. This experiment directly demonstrates the effectiveness of WAS.

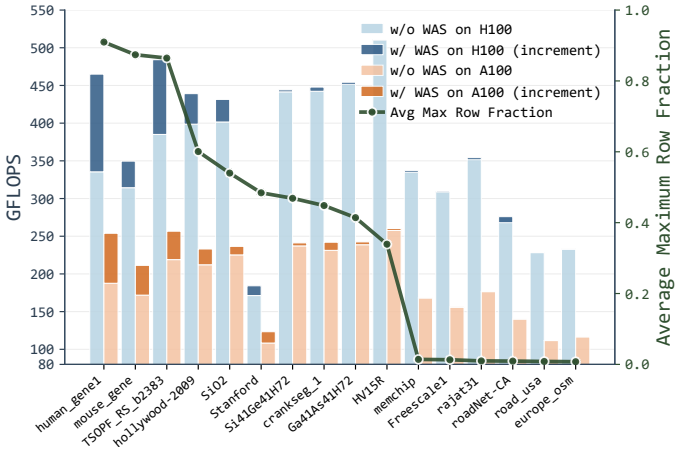
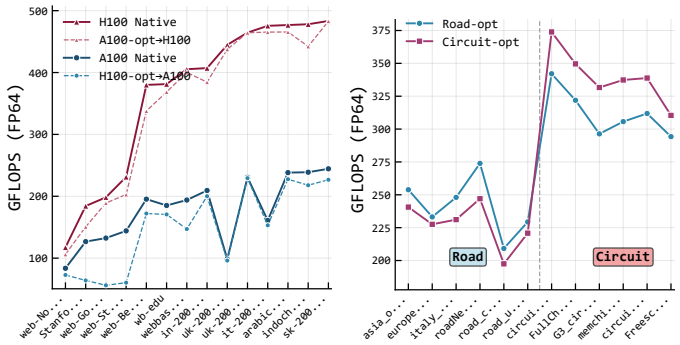


Fig. 15: WAS effectiveness ablation test.



(a) Arch-opt code performance (b) Domain-opt performance

Fig. 16: ACG effectiveness ablation test.

3) Architecture-Domain Co-Aware Analysis: In the generated kernels, we set the number of non-zeros per tile ($Tile_{\text{nnz}}$) to $\kappa \times Blk_{\text{thr}}$, assigning κ non-zeros to each thread. Fig. 16(a) illustrates the performance degradation when A100-optimized kernels run on an H100 GPU. On the A100, the optimal configuration ($Blk_{\text{thr}} = 256, \kappa = 4$) relies on high thread concurrency to hide memory latency. Conversely, the H100 favors $Blk_{\text{thr}} = 128$ and $\kappa = 8$. This setup increases the per-thread workload, thereby enhancing instruction-level parallelism (ILP). The H100's larger L2 cache and superior HBM bandwidth easily sustain this expanded working set without inducing cache thrashing. Consequently, applying the A100's high-concurrency, low-ILP parameters to the H100 underutilizes its enhanced memory subsystem. This confirms that ACG systematically captures generational microarchitectural shifts. Fig. 16(b) shows that kernels optimized for one domain perform poorly on others on the H100. These results validate ACG's effectiveness in adapting to both specific hardware architectures and distinct application domains.

VIII. Related Work

A. Conventional Preprocessing-Centric SpMV

In recent years, numerous SpMV optimizations have emerged, primarily focusing on memory access locality [3], [15], [22], [32]–[37], load balancing [38]–[43], and autotuning [6], [16]–[18], [23], [44]–[47]. Building upon the CSR format, enhanced preprocessing methods encompass HOLA-SpMV [7], CVR [21], CSR5 [5], FLAT [48], CSR-Adaptive [12], CSR2 [49], PCSR [50], ACSR [51], CSX [52], and CSR-Stream [53]. Among these, CSR5, HOLA-SpMV are representative CSR-based nnz-split algorithms that partition non-zeros into equal-workload tiles. CSR5 represents a classic preprocessing approach, performing comprehensive format conversion, including data transposition and generation of complex tile-specific metadata. These methods rely on high iteration counts to amortize preprocessing costs, limiting their effectiveness in low-iteration applications.

B. Preprocessing Optimization

Few works target matrix preprocessing optimization, with *GPUs All Grown-Up* [28] being a notable effort. It leverages AMD's *Command Processor* and the *Work Graphs* API to fuse preprocessing and computation into a single device-driven graph execution. While this reduces initial preprocessing overhead, it is only effective for low-iteration scenarios: Its best variant is outperformed by conventional amortized methods (CSR-Adaptive) after ∼92 iterations. This limitation stems from reduced kernel efficiency when fusing preprocessing, while conventional methods can amortize preprocessing overhead. SPIKe's strategy is fundamentally different, as it combines offline generation with runtime adaptation. This generation-adaptation duality sidesteps the fusion compromise, enabling high-efficiency execution across a wide spectrum of iteration counts, in contrast to the limited effective range of fusion-based methods.

IX. Conclusion

This paper presents SPIKe, a sparse iterative kernel generation method, which generates SpMV kernels that can be efficiently applied in sparse iterative methods with a wide range of iterations. SPIKe introduces an offline-generation, online-adaptation method, generating high-performance domain-sparsity-driven kernels while minimizing its preprocessing overhead. By combining online raw-data-driven adaptation with offline domain-specific code generation, SPIKe liberates preprocessing-intensive sparse matrix computations from the high-iteration constraint. We believe this paradigm extends naturally to other sparse operators. A current limitation is SPIKe's reliance on matrix collection's dataset labels for domain categorization, which requires end-users to have prior knowledge of their application context. Future work will introduce structural profiling to autonomously match unannotated matrices to pre-compiled kernels based on sparsity similarity.

Acknowledgment

The authors sincerely thank the anonymous reviewers for their insightful and constructive comments, which have greatly helped improve the quality of this paper. This work is supported by National Key Research & Development Program of China under Grant 2023YFB3001700 and National Natural Science Foundation of China under Grant 62372432.

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2003. [Online]. Available: https://doi.org/10.1137/1.9780898718003

[2] G. M. Del Corso, A. Gulli, and F. Romani, "Fast pagerank computation via a sparse linear system," *Internet Mathematics*, vol. 2, no. 3, pp. 251–273, 2005. [Online]. Available: https://doi.org/10.1007/978-3-540-30216-2_10

[3] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding up spmv for power-law graph analytics by enhancing locality & vectorization," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00090

[4] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: yet another spmv framework on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 107–118. [Online]. Available: https://doi.org/10.1145/2555243.2555255

[5] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350. [Online]. Available: https://doi.org/10.1145/2751205.2751209

[6] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 117–126. [Online]. Available: https://doi.org/10.1145/2491956.2462181

[7] M. Steinberger, R. Zayer, and H.-P. Seidel, "Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3079079.3079086

[8] NVIDIA Corporation, "cusparse library documentation," https://docs.nvidia.com/cuda/cusparse/, 2025, accessed on Aug 3, 2025 Version 12.9.

[9] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2026, version 0.6.0. [Online]. Available: https://github.com/cusplibrary/cusplibrary

[10] Y. Liu and B. Schmidt, "Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 82–89. [Online]. Available: https://doi.org/10.1109/ASAP.2015.7245713

[11] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 678–689. [Online]. Available: https://doi.org/10.1109/SC.2016.57

[12] M. Daga and J. L. Greathouse, "Structural agnostic spmv: Adapting csr-adaptive for irregular matrices," in *2015 IEEE 22nd International conference on high performance computing (HiPC)*. IEEE, 2015, pp. 64–74. [Online]. Available: https://doi.org/10.1109/HiPC.2015.55

[13] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web page*, Argonne National Laboratory, 2019. [Online]. Available: http://www.mcs.anl.gov/petsc

[14] T. A. Davis, "Algorithm1037: Suitesparse:graphblas: Parallel graph algorithms in the language of sparse linear algebra," vol. 49, no. 3, Sep. 2023. [Online]. Available: https://doi.org/10.1145/3577195

[15] N. Namashivayam, S. Mehta, and P.-C. Yew, "Variable-sized blocks for locality-aware spmv," ser. CGO '21. IEEE Press, 2021, p. 211–221. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370327

[16] G. Tan, J. Liu, and J. Li, "Design and implementation of adaptive spmv library for multicore and many-core architecture," *ACM Trans. Math. Softw.*, vol. 44, no. 4, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3218823

[17] A. Elafrou, G. Goumas, and N. Koziris, "Basmat: bottleneck-aware sparse matrix-vector multiplication auto-tuning on gpgpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 423–424. [Online]. Available: https://doi.org/10.1145/3293883.3301490

[18] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Wise: Predicting the performance of sparse matrix vector multiplication with machine learning," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 329–341. [Online]. Available: https://doi.org/10.1145/3572848.3577506

[19] I. R. eguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based gpus," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–12. [Online]. Available: https://doi.org/10.1109/InPar.2012.6339602

[20] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11. [Online]. Available: https://doi.org/10.1145/1654059.1654078

[21] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: efficient vectorization of spmv on x86 processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 149–162. [Online]. Available: https://doi.org/10.1145/3168818

[22] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926278

[23] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 94–108. [Online]. Available: https://doi.org/10.1145/3178487.3178495

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[25] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012. [Online]. Available: https://doi.org/10.1016/j.parco.2011.09.001

[26] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. USA: IEEE Computer Society, 1998, p. 1–27. [Online]. Available: https://doi.org/10.1109/SC.1998.10004

[27] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. [Online]. Available: https://doi.org/10.1109/JPROC.2004.840301

[28] F. Wildgrube, P. Ehrett, P. Trojahn, R. Membarth, B. Beckmann, D. Baumeister, and M. Chajdas, "Gpus all grown-up: Fully device-driven spmv using gpu work graphs," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1777–1791. [Online]. Available: https://doi.org/10.1145/3695053.3731060

[29] Y. Lu and W. Liu, "Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607051

[30] K. Zhang, H. Yang, X. You, T. Feng, Y. Xu, Z. Luan, Y. Liu, and D. Qian, "Exploiting efficient mapping and pipelined execution for accelerating spmv on tensor cores," in *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 245–258. [Online]. Available: https://doi.org/10.1145/3774934.3786441

[31] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "Lagraph: A community effort to collect graph algorithms built on top of the graphblas," in *2019 IEEE International Parallel and Distributed Processing Symposium*

Workshops (IPDPSW), 2019, pp. 276–284. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00053

[32] A. Elafrou, G. Goumas, and N. Koziris, "Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors," in *2017 46th International Conference on Parallel Processing (ICPP)*, 2017, pp. 292–301. [Online]. Available: https://doi.org/10.1109/ICPP.2017.38

[33] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 273–282. [Online]. Available: https://doi.org/10.1145/2464996.2465013

[34] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "Sparsex: A library for high-performance sparse matrix-vector multiplication on multicore platforms," *ACM Trans. Math. Softw.*, vol. 44, no. 3, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3134442

[35] A.-J. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, 2014. [Online]. Available: https://doi.org/10.1109/TPDS.2013.31

[36] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356148

[37] E. Karimi, N. B. Agostini, S. Dong, and D. Kaeli, "Vcsr: An efficient gpu memory-aware sparse format," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3977–3989, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2022.3177291

[38] M. Garland, "Sparse matrix computations on manycore gpu's," in *2008 45th ACM/IEEE Design Automation Conference*, 2008, pp. 2–6. [Online]. Available: https://doi.org/10.1145/1391469.1391473

[39] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3380930

[40] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus: implications for graph mining," *Proc. VLDB Endow.*, vol. 4, no. 4, p. 231–242, Jan. 2011. [Online]. Available: https://doi.org/10.14778/1938545.1938548

[41] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned spmv on gpus and multicore cpus," *IEEE Trans. Comput.*, vol. 64, no. 9, p. 2623–2636, Sep. 2015. [Online]. Available: https://doi.org/10.1109/TC.2014.2366731

[42] J. Guo, R. Xia, J. Liu, X. Zhu, and X. Zhang, "Camlb-spmv: An efficient cache-aware memory load-balancing spmv on cpu," in *Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 640–649. [Online]. Available: https://doi.org/10.1145/3673038.3673042

[43] J. Gao, W. Ji, J. Liu, Y. Wang, and F. Shi, "Revisiting thread configuration of spmv kernels on gpu: A machine learning based approach," *J. Parallel Distrib. Comput.*, vol. 185, no. C, Mar. 2024. [Online]. Available: https://doi.org/10.1016/j.jpdc.2023.104799

[44] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 99–108. [Online]. Available: https://doi.org/10.1145/2751205.2751244

[45] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 496–505. [Online]. Available: https://doi.org/10.1109/ICPP.2016.64

[46] G. Xiao, T. Zhou, Y. Chen, Y. Hu, and K. Li, "Machine learning-based kernel selector for spmv optimization in graph analysis," *ACM Trans. Parallel Comput.*, vol. 11, no. 2, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3652579

[47] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 68–78. [Online]. Available: https://doi.org/10.1109/IPDPS49936.2021.00016

[48] G. Chu, Y. He, L. Dong, Z. Ding, D. Chen, H. Bai, X. Wang, and C. Hu, "Efficient algorithm design of optimizing spmv on gpu," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 115–128. [Online]. Available: https://doi.org/10.1145/3588195.3593002

[49] H. Bian, J. Huang, R. Dong, L. Liu, and X. Wang, "Csr2: A new format for simd-accelerated spmv," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 350–359. [Online]. Available: https://doi.org/10.1109/CCGrid49817.2020.00-58

[50] G. He and J. Gao, "A novel csr-based sparse matrix-vector multiplication on gpus," *Mathematical Problems in Engineering*, vol. 2016, 2016. [Online]. Available: https://doi.org/10.1155/2016/8471283

[51] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 781–792. [Online]. Available: https://doi.org/10.1109/SC.2014.69

[52] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: an extended compression format for spmv on shared memory systems," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 247–256. [Online]. Available: https://doi.org/10.1145/1941553.1941587

[53] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 769–780. [Online]. Available: https://doi.org/10.1109/SC.2014.68