SparStencil: Retargeting Sparse Tensor Cores to Scientific Stencil Computations via Structured Sparsity Transformation

Qi Li*
University of Science and Technology of China
Hefei, China
liqi123@mail.ustc.edu.cn

Kun Li†
Microsoft Research
Beijing, China
likungw@gmail.com

Haozhi Han*
School of Computer Science
Peking University
Beijing, China
haozhi.han@stu.pku.edu.cn

Liang Yuan
Chinee Academy of Sciences
Beijing, China
yuanliang@ict.ac.cn

Yunquan Zhang
University of Chinese Academy of Sciences
Beijing, China
zyq@ict.ac.cn

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

Junshi Chen
University of Science and Technology of China
Hefei, China
cjuns@ustc.edu.cn

Hong An
University of Science and Technology of China
Hefei, China
han@ustc.edu.cn

Ting Cao
Tsinghua University
Beijing, China
tingcao@mail.tsinghua.edu.cn

Mao Yang
Microsoft Research
Beijing, China
maoyang@microsoft.com

Abstract

Sparse Tensor Cores offer exceptional performance gains for AI workloads by exploiting structured 2:4 sparsity. However, their potential remains untapped for core scientific workloads such as stencil computations, which exhibit irregular sparsity patterns.

This paper presents SparStencil, the first system to retarget sparse TCUs for scientific stencil computations through structured sparsity transformation. SparStencil introduces three key techniques: (1) Adaptive Layout Morphing, which restructures stencil patterns into staircase-aligned sparse matrices via a flatten-and-crush pipeline; (2) Structured Sparsity Conversion, which formulates transformation as a graph matching problem to ensure compatibility with 2:4 sparsity constraints; (3) Automatic Kernel Generation, which compiles transformed stencils into optimized sparse MMA kernels via layout search and table-driven memory mapping. Evaluated on 79 stencil kernels spanning diverse scientific domains, SparStencil achieves up to 7.1x speedup (3.1x on average) over state-of-the-art framework while reducing code complexity and matching or exceeding expert-tuned performance in both compute throughput and memory efficiency.

CCS Concepts

• **Computing methodologies** → **Parallel algorithms**; • **Computer systems organization** → Parallel architectures.

Keywords

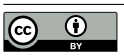
Stencil Computation, Structured Sparsity, Matrix Multiplication, Sparse Tensor Cores

ACM Reference Format:
Qi Li, Kun Li, Haozhi Han, Liang Yuan, Yunquan Zhang, Yifeng Chen, Junshi Chen, Hong An, Ting Cao, and Mao Yang. 2025. SparStencil: Retargeting Sparse Tensor Cores to Scientific Stencil Computations via Structured Sparsity Transformation. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3712285.3759820

1 Introduction

The rapid ascent of artificial intelligence (AI) workloads has fundamentally reshaped the computing landscape. Modern accelerators—especially GPUs—are now engineered not for general-purpose computing, but to saturate the needs of AI-native operations, such as matrix multiplications with structured sparsity. At the heart of this shift are Sparse Tensor Cores, which exploit fine-grained sparsity patterns (e.g., 2:4 sparsity) to skip redundant zero-element computations and significantly boost performance.

*Work done during an internship at Microsoft Research (May 1, 2024)
†Corresponding author: likungw@gmail.com



SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759820

Yet, scientific computing has largely remained on the periphery of this acceleration revolution. While sparse TCUs have become a cornerstone of AI performance, their capabilities remain underutilized in scientific applications, where data and compute structures diverge sharply from those in AI. This disconnect reflects a deeper tension: hardware is evolving toward AI-native execution models, but science remains anchored in patterns that predate them.

Nowhere is this divergence more evident than in stencil computations, the backbone of applications such as fluid dynamics [20, 25], weather prediction [1, 5], and earth modeling [21]. These computations iteratively update grid values based on local neighborhoods and account for over 75% of the runtime in long-running simulations [11]. Their ubiquity and cost have earned them recognition as one of the "seven dwarfs" of scientific computing [2, 3, 23].

However, stencil computations yield clustered, irregular sparsity patterns that fundamentally differ from the dense or uniformly sparse structures seen in AI. When mapped into matrix formats for GPU acceleration, these patterns often lead to dense masking, padding, and inefficient memory access—all of which violate the alignment constraints of sparse TCUs and degrade utilization. Prior approaches have largely focused on suppressing this sparsity through dense emulation, missing the opportunity to align it with the execution model of sparse TCUs.

This paper asks a fundamental question: Can we reimagine scientific computing not as a victim of hardware divergence but as a domain whose latent structure can be transformed to align with AI-native accelerators?

We argue yes. And we answer with SparStencil, a stencil computing system that retargets sparse tensor cores originally designed for AI workloads to accelerate scientific stencil computations. Instead of coercing stencil computations into dense formats, SparStencil embraces their clustered sparsity, restructures it into 2:4-compatible formats, and compiles it into optimized sparse TCU kernels—automatically and portably.

SparStencil is guided by four key insights that reshape how stencil sparsity can be efficiently exploited on sparse tensor cores. (1) Sparsity is not overhead—it's alignment potential. Prior approaches treated stencil-induced sparsity as overhead to be suppressed in order to fit dense compute units. Instead, we view this sparsity as an alignment opportunity for specialized sparse execution. (2) Residual sparsity is a leverage point. Despite masking and padding, stencil workloads retain high residual sparsity (50–80%). These conditions are poorly handled by dense TCUs but can be harnessed by sparse TCUs once the structure is exposed. (3) The irregularity does not mean a lack of structure. Though irregular at first glance, stencil-induced sparsity exhibits consistently clustered spatial patterns. These can be systematically reshaped into alignment-friendly forms with the right abstraction and transformation. (4) Structure-aware compilation closes the loop. Exploiting sparse TCUs demands more than format conversion—it requires compilers that understand and reason over structure, automating the path to performance.

To realize these insights, SparStencil consists of three tightly integrated stages:

First, Adaptive Layout Morphing reformulates diverse stencil patterns into unified matrix-matrix representations through a two-step flatten-and-crush pipeline. It systematically eliminates redundant data accesses while exposing staircase-aligned sparsity, a regularized layout pattern that serves as the structural backbone for downstream hardware-oriented transformations.

Second, Structured Sparsity Conversion transforms staircase-aligned sparsity into 2:4-compatible formats required by sparse TCUs. It is achieved by formulating a column-wise conflict graph, where nodes represent matrix columns and edges encode alignment violations. A hierarchical two-level matching algorithm is then applied to resolve conflicts, ensuring valid 2:4 alignment while minimizing zero-padding and maximizing TCU utilization.

Third, Automatic Kernel Generation compiles the transformed computation into high-performance sparse MMA kernels. It integrates layout exploration, table-driven memory mapping, and code synthesis into a fully automated backend, delivering expert-level performance without manual tuning or architecture-specific reengineering.

We evaluate SparStencil across 79 real-world stencil kernels drawn from a wide spectrum of application domains, including partial differential equation solvers, fluid dynamics, lattice Boltzmann methods, phase field models, and geophysical simulations. Compared to state-of-the-art baselines, SparStencil achieves an average 3.1x speedup, peaking at 7.1x, while significantly reducing code complexity. Our automatic compiler approach matches or exceeds manually optimized expert code in both compute throughput and memory efficiency.

To our knowledge, SparStencil is the first system to map scientific stencil computations onto sparse AI accelerators—not by tuning low-level kernels but by structurally rethinking how stencil workloads are represented and executed. This transformation retargets sparse Tensor Cores to scientific domains, and lays the groundwork for a broader shift toward accelerator-centric design for scientific computing.

2 Background and Challenges

2.1 Sparse Tensor Core

Sparse Tensor Core is a cutting-edge hardware architecture specifically designed to accelerate sparse matrix multiplication, as described in Equation 1.

$$D_{m\times n} = (A_{m\times k} \odot M_{m\times k}) \times B_{k\times n} + C_{m\times n} \tag{1}$$

where $M_{m\times k}$ is a binary mask matrix in which each group of 4 consecutive elements in a row contains exactly 2 elements equal to 1 and 2 elements equal to 0. This constraint, referred to as 2:4 sparsity, is represented as follows:

$$M_{i,4j+l-4} \in \{0, 1\} \quad \text{and} \quad \sum_{l=1}^{4} M_{i,4j+l-4} = 2 \tag{2}$$

While sparse TCUs strictly enforce this 2:4 sparsity pattern, sub-2:4 patterns (0:4 and 1:4) can still be processed. This is achieved by treating one or two elements in the 4-element block as nonzero, even if they are zero. Since multiplying by zero does not affect the result, this approach preserves computational correctness.

At the hardware level, there is an additional constraint; sparse TCUs partition matrices into uniformly sized fragments for computation. Regardless of matrix size variations, these fragments remain fixed (e.g., $16 \times 16 \times 8$, $16 \times 32 \times 8$). [33]

Despite the stringent 2:4 sparsity constraints and invariant fragment size limitations imposed by sparse TCUs, it offers substantial computational potential. Across all supported data types (FP16, BF16, and TF32), sparse TCUs achieve up to 32x speedup over traditional Fused Multiply-Add (FFMA) operations and consistently deliver 2x the performance of dense TCUs, presenting a substantial opportunity to accelerate fundamental operators in traditional scientific computing domains, such as stencil computations. [32]

2.2 Stencil Computation

Stencil computations play a critical role in numerous scientific and engineering applications, which are extensively involved in various domains from physical simulations to machine learning. Stencil is also included as one of the seven computational dwarfs presented in the Berkeley View and arises as a principal class of floating-point kernels in high-performance computing.

A stencil contains a pre-defined pattern that updates each point in a d-dimensional spatial grid iteratively along the time dimension. These patterns are typically categorized into two main types: star and box stencils. In a star stencil, the computation involves a weighted sum of the central point and its neighboring points along individual dimensions. Conversely, a box stencil computes the weighted sum over a neighborhood that forms a square or cube around the central point in 2D or 3D, respectively.

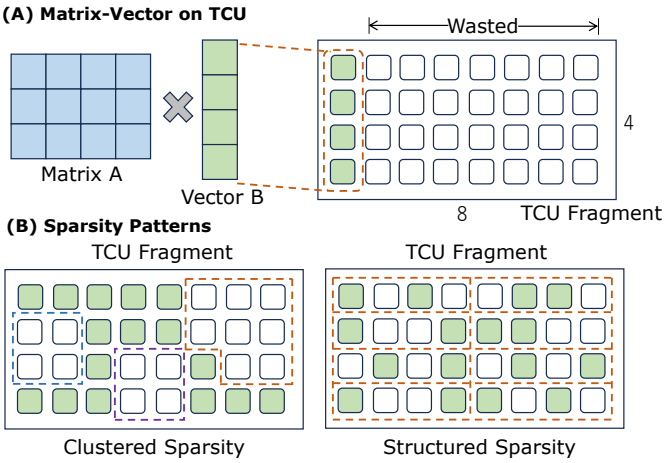


Figure 1: Architectural mapping challenges of stencil computations on TCUs. (a) Naive matrix-vector mapping achieves only 12.5% compute utilization ($\frac{1}{8}$ active columns) on an 8×4 TCU fragment due to rigid dataflow constraints. (b) The clustered sparsity and structured sparsity in TCUs.

2.3 Divergence and Opportunities

Sparse TCUs have provided dedicated acceleration for the most fundamental matrix multiplication operations in the AI field. However, essential scientific operators such as stencil, face significant challenges in effectively harnessing AI-specific accelerators like TCUs, despite its immense computational potential being a highly valuable yet underutilized resource.

For instance, a direct translation of stencil computations to naive matrix-vector multiplication via a canonical im2row-style approach exhibits severely low accelerator utilization. As shown in Figure 1(a), 87.5% columns are wasted due to the fixed fragment dimensions inherent in TCU architectures.

Fortunately, some efforts have already been made to bridge this divergence by mapping stencil computations onto dense TCUs, such as TCStencil and ConvStencil. However, these adaptations inevitably introduce more than 50% *clustered sparsity*, leading to the underutilization of more than half of the TCU's computational throughput. The *clustered sparsity*, as illustrated in Figure 1(b), refers to sparsity patterns containing contiguous regions with excessive zero concentrations that violate the structured sparsity requirements (2:4 sparsity) of sparse TCUs.

While clustered sparsity has traditionally been seen as a barrier to dense TCUs, it offers a unique opportunity in the context of sparse TCUs. If we can convert the clustered sparsity of the matrix into structured 2:4 sparsity, we can seamlessly integrate sparse TCUs into stencil computations. Instead of focusing on eliminating zero elements, this approach enables us to leverage the hardware to significantly increase computational density. Consequently, rather than viewing sparsity as a limitation, we propose that it should be seen as an opportunity to harness the capabilities of sparse TCUs, unlocking new pathways for stencil computations.

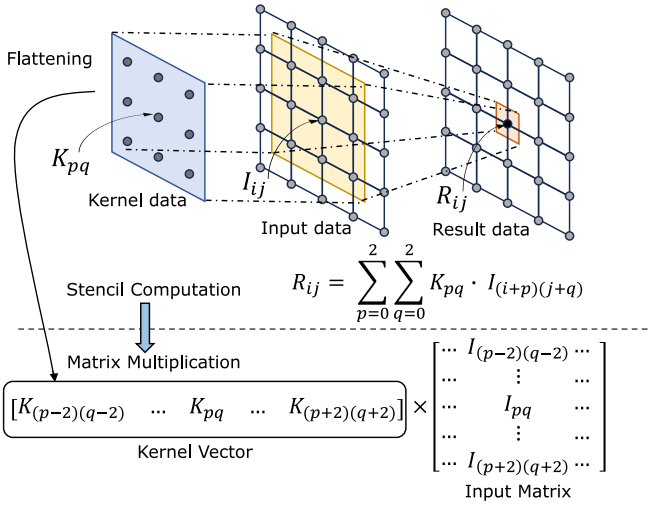


Figure 2: Example of the stencil flattening process.

2.4 Challenges

Despite the opportunity, systematically and efficiently mapping stencil computations with various stencil kernels onto sparse TCUs is not a straightforward task. In this subsection, we outline and discuss three key challenges:

Firstly, *how can we map stencil kernels onto TCUs with low redundancy and sparsity-awareness?* The matrix-based transformation of stencil computations inherently introduces extensive duplication, as sliding-kernel overlap causes each input element to appear repeatedly across the matrix, leading to significant memory overhead.

Moreover, existing works (e.g., TCStencil and ConvStencil) lack sparsity awareness, resulting in irregular patterns that cannot be aligned with sparse TCUs. Lacking the ability to restructure such sparsity into a usable form, they often choose to ignore it entirely and fall back to dense computation.

Secondly, *how can we convert sparsity into the fine-grained structured patterns required by sparse TCUs?* Sparse TCUs demand element-level alignment: every row must contain exactly two non-zeros within each four-element group. Yet stencil sparsity emerges from coarse-grained, spatially entangled structures that defy such regularity. This mismatch in granularity makes element-wise adjustment infeasible—nonzeros cannot simply be shuffled without corrupting stencil semantics. At the same time, coarse layout modifications—such as reordering rows or columns—often introduce cascading disruptions, where changes in one region break alignments elsewhere. As a result, even small transformations risk destabilizing the entire sparsity layout.

Thirdly, *how can we automatically generate highly optimized CUDA kernels with optimal layout mapping?* Stencil kernels admit a vast space of layout-to-execution mappings, each inducing distinct sparsity patterns, memory traffic, and alignment constraints. This space is too large and irregular for manual tuning, forcing existing TCU-based methods to rely on hand-tuned schedules tailored to specific kernel sizes. Moreover, the intricate interplay between sparsity patterns, TCU constraints, and kernel performance metrics also presents a significant challenge in defining the search space for identifying the optimal layouts that maximize TCU utilization.

3 SparStencil

In this section, we will provide a detailed introduction to SparStencil. It is a novel approach that leverages sparse TCUs for stencil computations for the first time, extending the boundaries of sparse TCUs for stencil computation.

3.1 Adaptive Layout Morphing

Adaptive Layout Morphing is designed with two fundamental objectives: (1) to reformulate stencil computations into matrix representations flexibly, and (2) to implement a sparsity-aware strategy that refines clustered sparsity into a more structured, staircase-like sparsity pattern while consolidating consecutive redundant elements. These objectives are achieved through *Stencil Flattening* and *Duplications Crush*, respectively.

Stencil Flattening generalizes the transformation of stencil computations—originally incompatible with TCU—into a unified vector-matrix multiplication format suitable for TCUs. Specifically, the approach unfolds the stencil kernel weights into a single-row vector, the *kernel vector*, while simultaneously reshaping each region of the input data covered by the sliding kernel into corresponding column vectors. Concatenating these column vectors row-wise forms the *input matrix*. Consequently, stencil computations, previously executed as repeated weighted summations, become a vector-matrix multiplication. Figure 2 illustrates the stencil flattening process for a 3×3 kernel applied to a 5×5 input matrix.

However, directly applying stencil flattening to TCUs often leads to low hardware utilization, as the kernel vector occupies only one row of the TCU fragment, leaving other rows unused. Moreover, this transformation inherently generates significant redundancy: the input matrix contains numerous duplicated elements, aggravating memory-bound constraints inherent in stencil computations.

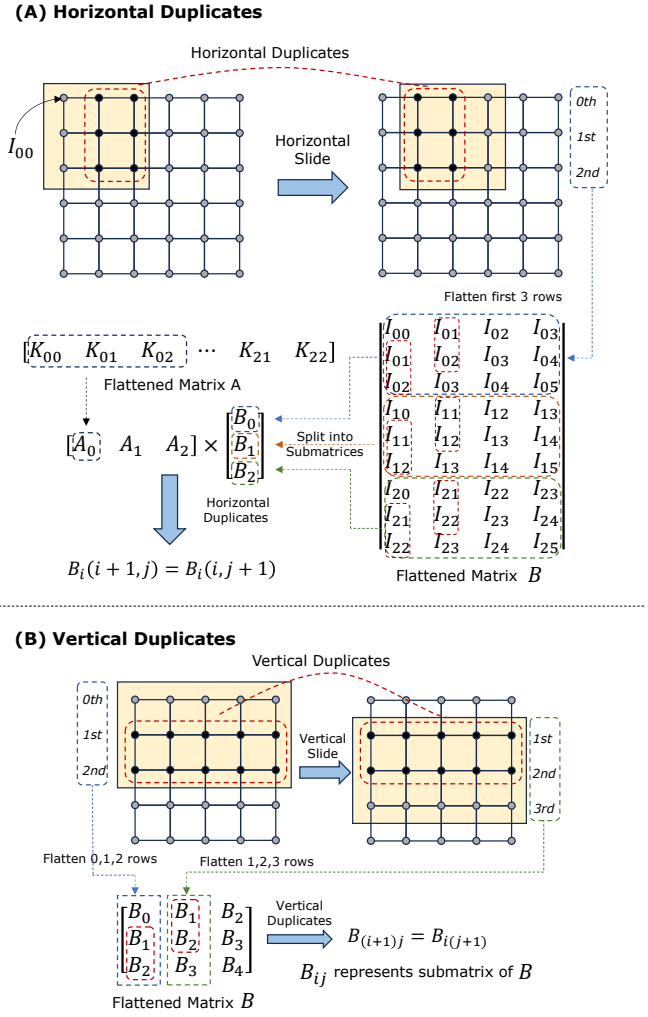


Figure 3: The distribution of duplicates after stencil flattening. (a) The horizontal duplicate distribution at the element level within a sub-matrix. (b) The vertical duplicates distribution between sub-matrices at the sub-matrix level.

To address these issues, we introduce *Duplicates Crush*.[1] By crushing redundant elements within the input matrix, it adaptively transforms the *vector-matrix* multiplication into *matrix-matrix* multiplication according to the TCU fragment size while intentionally guiding a staircase-like sparsity pattern, laying the groundwork for the subsequent transformation. The *Duplicates* refers to the element duplicates caused by the horizontal and vertical sliding of the kernel in stencil computations, which can be categorized along the two directions into two types: horizontal duplicates and vertical duplicates.

[1]The inspiration for Duplicates Crush comes from the game Candy Crush, where elements are eliminated in a similar manner. Here, we analogously eliminate duplicates.

The horizontal duplicates are illustrated in Figure 3(a). During horizontal sliding, the stencil kernel repeatedly covers certain elements due to overlapping regions at adjacent positions. These repeated elements are reflected in matrix B' after Stencil Flattening, such as I_{01} and I_{02}, I_{11} and I_{12}, as well as I_{21} and I_{22}. In Figure 3(a), B' is partitioned into three submatrices, where each B_i consists solely of elements from the i-th input row. Due to horizontal kernel sliding, adjacent kernels in Figure 3(a) share two elements per row. This overlap induces horizontal duplicates in B, with adjacent columns in each B_i sharing two identical elements. Generally, these horizontal duplicates can be mathematically expressed as:

$$B_i(i+1, j) = B_i(i, j+1) \tag{3}$$

where $B_i \in \mathbb{R}^{k\times(n-k+1)}$, $B \in \mathbb{R}^{k^2\times(m-k+1)(n-k+1)}$ and B_i is a submatrix of B.

The vertical duplicates are caused by the vertical sliding of the stencil kernel. As illustrated in Figure 3(b), elements from the first row and the second row are covered simultaneously by these adjacent kernels. After flattening, these repeated elements appear as submatrices in B', where the elements from the first row become submatrix B_1 and the elements from the second row become submatrix B_2. Viewing the matrix B' as consisting of submatrix B_i, each composed of the elements from the i-th row, the vertical redundancy in B' manifests as duplication between these submatrices. Mathematically, this can be expressed as:

$$B'_{i+1,j} = B'_{i,j+1} \tag{4}$$

where $B'_{i,j}$ denotes the element located at the (i, j) position within the submatrix of B'.

After analyzing the distribution of duplicates, we now introduce the *Duplicates Crush*. It eliminates the duplicates in matrix B' after *Stencil Flattening*, while expanding the vector A into a matrix A' and intentionally introducing a staircase-like sparsity pattern within A to prepare for the transformation. Similar to how duplicates are distributed, the crushing procedure is categorized along the stencil kernel's two sliding directions: horizontal and vertical.

We begin with horizontal duplicate crushing, as characterized in Equation 3. Since these duplicates are confined within each submatrix B_i of B', it suffices to illustrate the process at the sub-matrix level for uniform application across all B_i. The procedure, illustrated in Figure 4(a), proceeds as follows: (1) Consider the flattened matrices A' and B' in sub-matrix form, and select A_0 and B_0 as representative examples. (2) In B_0, duplicate values appear diagonally across columns. These are eliminated by retaining one representative per group and reorganizing the unique entries into a single-column format. (3) Concurrently, A_0 is vertically expanded into multiple rows with deliberate zero-padding to align with the new structure, forming a staircase-like sparsity pattern. Although not directly compatible with sparse TCUs, the staircase pattern imposes a regular layout that facilitates the subsequent transformation.

We next describe vertical duplicate crushing, which targets duplicates across sub-matrices in B', as defined in Equation 4. The crushing process is again applied at the sub-matrix level, as illustrated in Figure 4(b), and proceeds as follows: (1) After the horizontal duplicate crushing, vertical duplicates in B' exhibit diagonal alignment at the sub-matrix level, similar to the horizontal case.

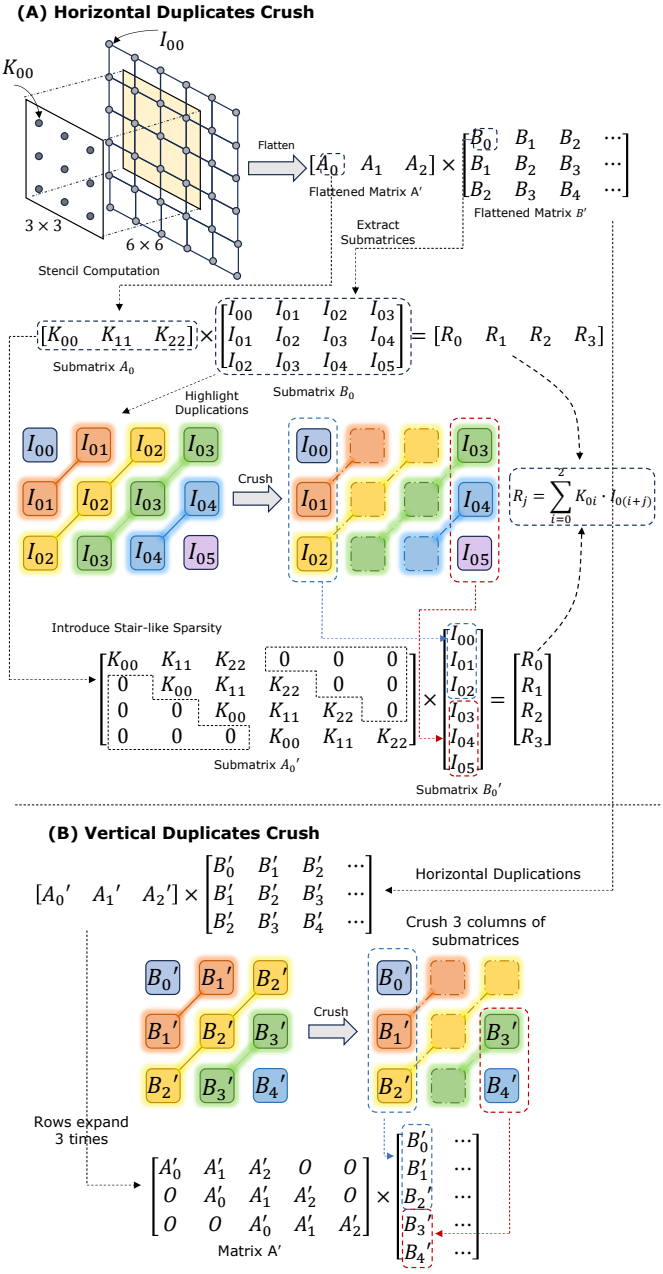


Figure 4: The process of Duplicates Crush. Figure (a) illustrates the horizontal duplicates crush at the element level within a sub-matrix. Figure (b) shows the vertical duplicates crush between sub-matrices at the sub-matrix level.

(2) Redundant sub-matrices across the three columns are removed, retaining one representative per group. The remaining sub-matrices are then reorganized into a single column structure. (3) Meanwhile, A is similarly expanded into multiple rows at the sub-matrix level.

The above procedures illustrate both horizontal and vertical duplicate crushing. In general, horizontal crushing can be applied to every r_1 columns within each B_i, while vertical crushing merges

every r_2 columns of B'. The values of r_1 and r_2 are determined during the subsequent layout exploration phase.

3.2 Structured Sparsity Conversion

Although *Adaptive Layout Morphing* guides sparsity into a staircase-like structure, the resulting pattern remains irregular and violates the 2:4 constraint required by sparse TCUs. To bridge this gap, we propose *Structured Sparsity Conversion*, a general transformation that reorganizes intermediate sparsity into a hardware-aligned 2:4 format.

A core component of this process is the *Permutation Invariant Transformation (PIT)*, which preserves computation semantics during layout reordering. PIT simultaneously permutes the columns of $\mathbf{A}$ and the corresponding rows of $\mathbf{B}$ along the k-dimension, ensuring the result of $\mathbf{A} \times \mathbf{B}$ remains unchanged. Formally, let $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_k] \in \mathbb{R}^{m \times k}$ and $\mathbf{B} = [\mathbf{b}_1^\top; \ldots; \mathbf{b}_k^\top] \in \mathbb{R}^{k \times n}$. Then,

$$\mathbf{C} = \sum_{i=1}^{k} \mathbf{a}_i \mathbf{b}_i^\top = \sum_{i=1}^{k} \mathbf{a}_{\mathcal{P}(i)} \mathbf{b}_{\mathcal{P}(i)}^\top, \tag{5}$$

guaranteeing that the multiplication remains invariant under any shared permutation $\mathcal{P}$—a crucial property for enabling valid sparse pattern adaptation.

While PIT guarantees correctness, not all permutations yield valid 2:4 structured sparsity. To make the search space tractable, we observe that the 2:4 constraint can be equivalently expressed as a composition of 1:2 and 0:2 subpatterns (Section 2.1). This motivates the core problem of *Structured Sparsity Conversion*: finding a permutation $\mathcal{P}$ that rearranges nonzeros into conflict-free 1:2 or 0:2 groupings while minimizing padding. We formalize this as follows:

Given a matrix $\mathbf{A}' \in \mathbb{R}^{m \times n}$ derived from layout morphing with parameters (r_1, r_2), we construct an augmented matrix $\mathbf{A}'_{\text{pad}} \in \mathbb{R}^{m \times (n+p)}$ by appending p zero columns. The goal is to find a permutation $\mathcal{P}$ such that the transformed matrix $\mathbf{A}'' = \mathcal{P}(\mathbf{A}')$ satisfies a 1:2 or 0:2 sparsity pattern while minimizing p. We model this as a graph-theoretic matching problem:

Definition 1 (Conflict Graph). *Given $\mathbf{A}' \in \mathbb{R}^{m \times n}$, its* conflict graph *is $G = (V, E)$, where $V = \{v_i\}_{i=1}^{n}$ represents the columns of $\mathbf{A}'$, and $(v_i, v_j) \in E$ if and only if $\exists r \in [m]$ such that $\mathbf{A}'[r, i] \neq 0$ and $\mathbf{A}'[r, j] \neq 0$.*

Definition 2 (Augmented Matching Graph). *Given zero columns $Z = \{z_1, \ldots, z_p\}$, the* augmented graph *is $G' = (V \cup Z, E)$, where E is inherited from the original conflict graph and no additional edges are introduced.*

Definition 3 (Valid Matching). *A matching $\mathcal{M} \subset \binom{V \cup Z}{2}$ is valid if it satisfies: (i)* coverage*: each $v \in V$ appears in exactly one edge in $\mathcal{M}$; and (ii)* conflict-freedom*: $\mathcal{M} \cap E = \emptyset$.*

Problem 1 (Minimum Zero-Column Matching). *Find the minimal $p \in \mathbb{N}$ such that a valid matching $\mathcal{M} \subset \binom{V \cup Z}{2}$ exists in $G' = (V \cup Z, E)$, with $|Z| = p$.*

To solve the problem, we introduce the following definition:

Definition 4 (k-Staircase Matrix). *A matrix $\mathbf{A}' \in \mathbb{R}^{m \times n}$ has the k-staircase property if:*

$$\mathbf{A}'[r, c] \neq 0 \iff r \leq c < r + k - 1 \quad (\forall r, c \in [n])$$

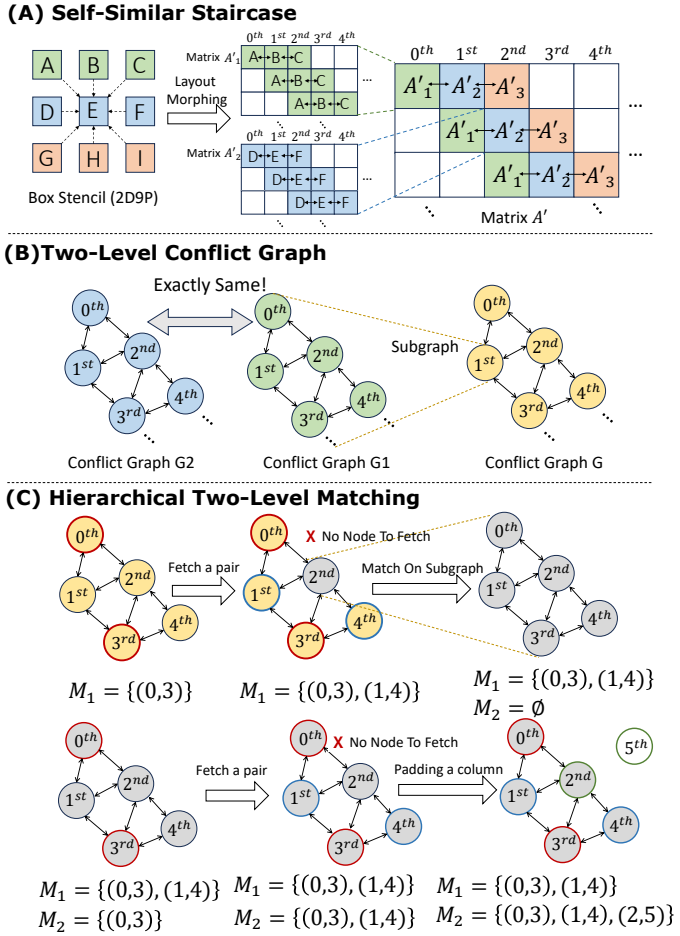


Figure 5: An example of Sparsity Conversion. (a) The self-similar staircase property in the matrix undergoing layout morphing. (b) The global conflict graph and the local conflict graph. (c) The process of hierarchical two-level matching algorithm.

where $[n] := \{0, 1, ..., n-1\}$ denotes indices.

As described in Section 3.1, the matrix $\mathbf{A}'$ can be partitioned into blocks $\{\mathbf{A}'_{i,j}\}$, exhibiting a self-similar k-staircase structure: (i) *Global Staircase*: treating each block $\mathbf{A}'_{i,j}$ as a scalar, the block-level matrix satisfies the k-staircase property; (ii) *Local Staircase*: each nonzero block $\mathbf{A}'_{i,j}$ individually satisfies the same property. Figure 5 (a) illustrates this self-similarity.

The self-similar structure of $\mathbf{A}'$ induces two levels of conflict graphs, as illustrated in Figure 5 (b): (i) *Global Conflict Graph* $G = (V, E)$: each node corresponds to a block column $\mathbf{A}'_{:,i}$, with $(i, j) \in E$ iff $\exists r$ such that $\mathbf{A}'_{r,i} \neq \mathbf{0}$ and $\mathbf{A}'_{r,j} \neq \mathbf{0}$; (ii) *Local Conflict Graph* $G_i = (V_i, E_i)$: each node corresponds to a column in $\mathbf{B}'_i$, with $(p, q) \in E_i$ iff $\exists r$ such that $\mathbf{A}'_{i,j}[r, p] \neq 0$ and $\mathbf{A}'_{i,j}[r, q] \neq 0$.

After constructing two-level conflict graphs, the following theorem can be derived based on the k-staircase property:

Theorem 1 (Non-Conflict Property). *Let $\mathbf{A}'$ be a matrix with a staircase pattern and its conflict graph $G = (V, E)$. Then any two nodes separated by at least k hops are* conflict-free*:*

$$\forall j \in [n-\delta], \quad (c_j, c_{j+\delta}) \notin E \quad \text{for all } \delta \geq k$$

Proof. Conflict free: By the staircase offset, columns c_j and $c_{j+\delta}$ satisfy:

$$\forall r, \mathbf{A}'[r, c_j] \neq 0 \implies \mathbf{A}'[r, c_{j+\delta}] = 0$$

Thus, no shared nonzero rows exist. □

With Theorem 1, we propose *Hierarchical Two-Level Matching* to solve the Problem 1, which is shown in Algorithm 1. The process of Algorithm 1 is shown in Figure 5(c). Furthermore, Theorem 2 establishes the correctness and optimality of this algorithm.

Theorem 2 (Correctness and Optimality). *Let $\mathcal{M}$ be the matching returned by Algorithm 1, and let Z be the set of inserted zero-columns. Then: (i)* Validity*: for all $(v_i, v_j) \in \mathcal{M}$, we have $|j - i| \geq k$; (ii)* Minimality*: $|Z| = p$ achieves the minimum value in Problem 1.*

Proof. (i) *Validity.* Let $(u, v) \in \mathcal{M}$. If $(G_i, G_j) \in \mathcal{M}_1$, then $j - i = s_1 \geq k$; if $(v_i, v_j) \in \mathcal{M}_2$, then $j - i = s_2 \geq k$. Hence, all matched pairs satisfy $|j - i| \geq k$. (ii) *Minimality.* Since all subgraphs are structurally identical, it suffices to analyze a single subgraph G_i of size g. If $k \leq \lfloor g/2 \rfloor$, then: when g is even, a perfect matching exists; when odd, exactly one node remains unmatched—this is trivially optimal. If $k > \lfloor g/2 \rfloor$, the algorithm pairs $v_1, \ldots, v_{g-k}$ with $v_{k+1}, \ldots, v_g$, matching $g - k$ nodes. Assume a better matching $\mathcal{M}'$ exists. Validity requires $j \geq i + k \Rightarrow i \leq g - k$ for any pair $(i, j) \in \mathcal{M}'$, so no pair with $i > g - k$ can exist, contradicting the assumption. Thus, the matching is optimal. □

Theorem 2 ensures that Algorithm 1 achieves both correctness and optimality with linear complexity $O(|V|)$, since each vertex is traversed at most once. While this is guaranteed under the k-staircase assumption, our approach remains broadly applicable even when the input deviates from this structure. In such cases, the same conflict graph formulation allows us to fall back to the classical Blossom algorithm [12] to compute maximum matchings over arbitrary sparsity patterns. Despite its worst-case complexity of $O(|E||V|^2)$, the Blossom algorithm remains practical in our setting, as real-world stencil kernels typically yield small and sparse conflict graphs.

3.3 Automatic Kernel Generation

Despite methods for leveraging sparse TCUs, manual tuning for sparse TCUs is both labor-intensive and inherently suboptimal due to the high variability of stencil kernels and TCU fragments. To address this issue, we propose *Automatic Kernel Generation*, a compiler framework that enables end-to-end scheduling through two phases: (1) *layout exploration* dynamically determines the optimal data layout by automatically searching over a tunable space, minimizing memory transfer overhead while maximizing computational density. (2) *Code Generation* emits expert-level CUDA kernels via metadata-driven sparse MMA, lookup-based memory mapping, and double-buffered pipelines, consistently outperforming hand-tuned implementations.

Algorithm 1: Hierarchical Two-Level Matching

Input: Global graph G with n nodes, subgraph size g, stencil size k
Output: Perfect matching $\mathcal{M}$

Initialize $m \leftarrow n/g$, $s_1 \leftarrow \max(\lfloor m/2 \rfloor, k)$, $\mathcal{M}_1 \leftarrow \emptyset$;
for $i = 0$ **to** $m - 1$ **do**
  if G_i *unmatched and* $i + s_1 < m$ **then**
    $\mathcal{M}_1 \leftarrow \mathcal{M}_1 \cup \{(G_i, G_{i+s_1})\}$;
Initialize $\mathcal{M}_2 \leftarrow \emptyset$, $s_2 \leftarrow \max(\lfloor g/2 \rfloor, k)$;
for *each unmatched* G_x **do**
  for $u = 0$ **to** $|V(G_x)| - 1$ **do**
    $v \leftarrow u + s_2$;
    if v_x^u *unmatched* **then**
      if $v < |V(G_x)|$ **then**
        $\mathcal{M}_2 \leftarrow \mathcal{M}_2 \cup \{(v_x^u, v_x^v)\}$;
      else
        Add zero node ζ_v, $\mathcal{M}_2 \leftarrow \mathcal{M}_2 \cup \{(v_x^u, \zeta_v)\}$;
Initialize $\mathcal{M} \leftarrow \emptyset$;
for $(G_p, G_q) \in \mathcal{M}_1$, $t = 0$ **to** $g - 1$ **do**
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{(v_p^t, v_q^t)\}$;
$\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_2$;

Layout Exploration. The configuration parameters (r_1, r_2) in Section 3.1 modulate data layout patterns and induce clustered sparsity in TCU, where r_1 and r_2 denote the number of consecutive columns to be merged in B' and B. The configuration creates distinct memory-computation trade-offs that necessitate automated exploration to identify optimal configurations. To systematically evaluate these configurations, we adopt an analytical model tailored for stencil computations on TCU [9], shown in Equation 6, 7 and 8.

$$T = max(T_{compute}, T_{memory}) \tag{6}$$

$$T_{compute} = \frac{N_{MMA} \times CPI_{tcu}}{f N_{tcu}} \tag{7}$$

$$T_{memory} = max(\frac{data_R}{bw_G} + \frac{data_W}{bw_G}, \frac{data_{transW}}{bw_S} + \frac{data_{transR}}{bw_S}) \tag{8}$$

Compute Evaluation. As defined in Equation 7, $T_{compute}$ depends on N_{MMA}, with CPI_{tcu}, f, and N_{tcu} fixed by hardware. Given an $m \times n$ input, a $k \times k$ stencil kernel, and a TCU fragment of size $M \times K \times N$, applying (r_1, r_2) layout morphing yields $\mathbf{A}' \in \mathbb{R}^{m' \times k'}$, $\mathbf{B}' \in \mathbb{R}^{k' \times n'}$ with $m' = r_1 r_2$, $k' = (k + r_1 - 1)(k + r_2 - 1)$, and $n' = \frac{(m-k+1)(n-k+1)}{r_1 r_2}$. The corresponding MMA count is:

$$N_{MMA} = \left\lceil \frac{m'}{M} \right\rceil \left\lceil \frac{k'}{K} \right\rceil \left\lceil \frac{n'}{N} \right\rceil \tag{9}$$

Substituting N_{MMA} into Equation 7 yields $T_{compute}$.

Memory Evaluation. SparStencil stores the original input $\mathbf{I}$ and the pre-converted kernel matrix $\mathbf{A}'$ in global memory. As stencil kernels are typically small, the footprint of $\mathbf{A}'$ is negligible compared to $\mathbf{I}$, which remains layout-invariant and thus independent of (r_1, r_2). Consequently, $data_R$ and $data_W$ in Equation 8 are constant,

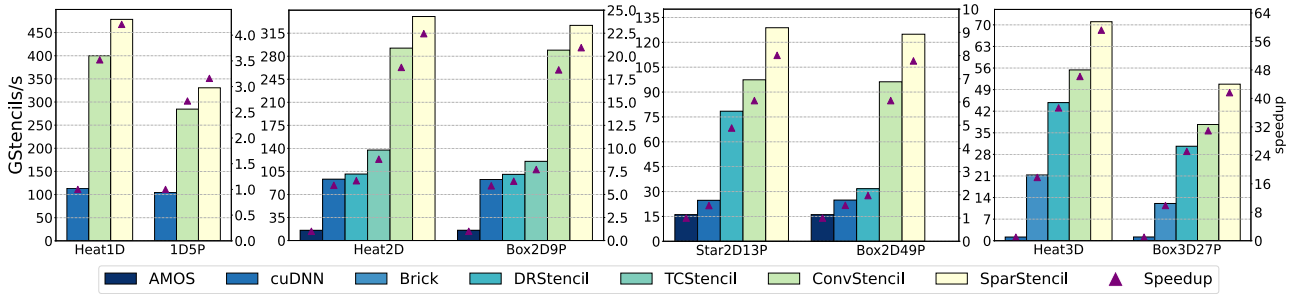


Figure 6: Performance comparison of SparStencil with state-of-the-arts.

while $data_{transR}$ and $data_{transW}$ depend on $\mathbf{A}'$ and $\mathbf{B}'$ after layout morphing:

$$data_{transR} = data_{transW} = k' \left(\frac{m}{2} + n' \right) \tag{10}$$

We compute T_{memory} via Equation 8, and total latency T using Equation 6. The optimal configuration is selected by exhaustive search:

$$(r_1^*, r_2^*) = \arg \min_{r_1, r_2 \in \mathcal{S}} T(r_1, r_2) \tag{11}$$

For each layout candidate (r_1, r_2), SparStencil conducts static graph analysis to construct a conflict graph from the symbolic sparsity of $\mathbf{A}'$. A provably optimal permutation is then derived via Algorithm 1 to satisfy the 2:4 sparsity constraint under structural constraints. The resulting permutation is applied via PIT, yielding a layout conforming to sparse TCU execution.

Code Generation. With optimal parameters (r_1, r_2) and the PIT transformation applied, the resulting irregular stencil patterns and hardware-specific layouts create highly scattered memory access patterns that are impractical to analyze manually. To address this, SparStencil automates CUDA kernel generation by integrating metadata encoding, lookup table acceleration, and a highly optimized CUDA template.

Metadata. Computing sparse matrix multiplication $\mathbf{A} \times \mathbf{B}$ on sparse TCUs necessitates the generation of metadata to indicate the positions of non-zero elements in matrix $\mathbf{A}$. The metadata generation process is straightforward: it involves traversing matrix $\mathbf{A}$, recording the indices of nonzero elements, and encoding them into a compact format.

Lookup Table. During layout transformation, computing address pointer offsets for transferring data from global to shared memory introduces significant computational overhead due to frequent integer division and modulus operations, which are inefficient on GPUs.

Table 1: Summary of symbols used in performance modeling.

Symbol	Description
T, T_{compute}, T_{memory}	Total, compute, and memory time
$data_{\text{transR}}$ / $data_{\text{transW}}$	Shared memory read/write volume
N_{MMA}, N_{tcu}	MMA operations number, TCUs number
$data_R$ / $data_W$	Global memory read/write volume
CPI_{tcu}, f	Cycles per TCU op, GPU core frequency
bw_G, bw_S	Bandwidth of global/shared memory

Moreover, these computations often exhibit redundancy across different processing blocks. To mitigate this, we precompute pointer offsets on the host and store them in lookup tables, which are then passed to the CUDA kernel. This approach eliminates redundant computations, significantly improving memory access efficiency.

Based on prior analysis, SparStencil instantiates optimized CUDA kernels from a predefined template tailored for memory efficiency, compute-transfer overlap, and sparse TCU utilization. The kernel follows a three-stage pipeline: (1) asynchronous loading via lookup tables to prefetch global memory into shared memory; (2) sparse MMA execution using metadata, interleaved with the next load to maximize throughput; and (3) writing results back to global memory, completing a double-buffered loop.

4 Evaluation

4.1 Setup

Implementation. We implement SparStencil with CUDA C++ and PTX. SparStencil is compiled with NVCC 12.4.

Machine. Our platform is composed of an AMD EPYC 7543 processor and an Nvidia A100 GPU with sparse Tensor Core. The A100 GPU is connected to the motherboard via PCIe Gen4. The A100 GPU is equipped with 108 SMs, each containing 4 sparse Tensor Cores [31].

State-of-the-arts. We compare SparStencil with a wide range of state-of-the-arts, including cuDNN, AMOS, Brick, DRStencil, TCStencil, and ConvStencil in FP16. Moreover, considering that ConvStencil employs 3x temporal fusion for small kernels, SparStencil adopts the same approach in our comparative evaluation to ensure a fair performance comparison.

Benchmarks. We use various stencil kernels with different shapes as benchmarks. The specifics are detailed in Table 2 [18, 46].

Metrics. Most works on stencil evaluate performance using GStencil/s [8, 28, 45–48] (Gigastencils per second, denoting the number of stencil points updated per second) as a metric. We also adopt this metric, as defined in Equation 12.

$$\text{GStencil/s} = \frac{T \times \prod_{i=1}^{n} N_i}{t \times 10^9} \tag{12}$$

where T denotes the number of iterations, n denotes the dimensionality of the stencil, N_i denotes the size of the i-th dimension, and t denotes the total execution time in seconds.

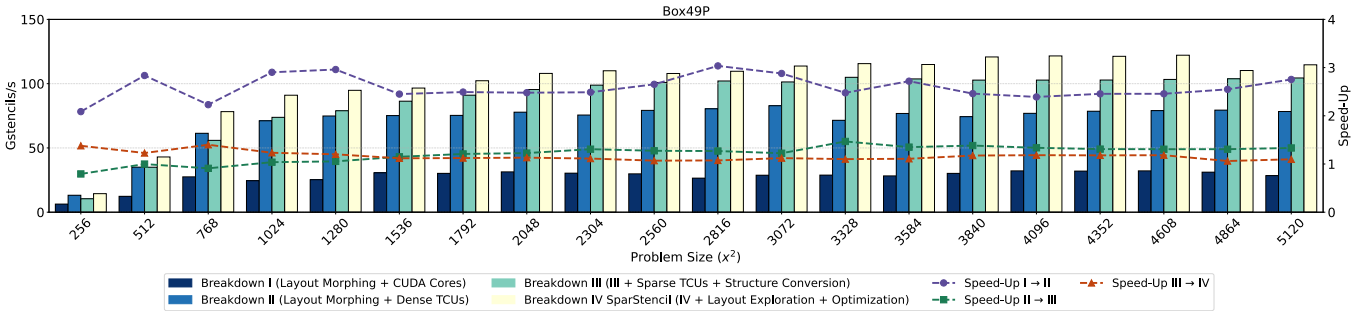


Figure 7: Performance breakdown of SparStencil

Table 2: Configuration for Stencil Benchmarks

Kernel	Points	Problem Size	Block
Heat-1D	3	10240000 × 10000	1024
1D5P	5	10240000 × 10000	1024
Heat-2D	5	10240 × 10240 × 10240	32 × 64
Box-2D9P	9	10240 × 10240 × 10240	32 × 64
Star-2D13P	13	10240 × 10240 × 10240	32 × 64
Box-2D49P	49	10240 × 10240 × 10240	32 × 64
Heat-3D	7	1024 × 1024 × 1024 × 1024	8 × 64
Box-3D27P	27	1024 × 1024 × 1024 × 1024	8 × 64

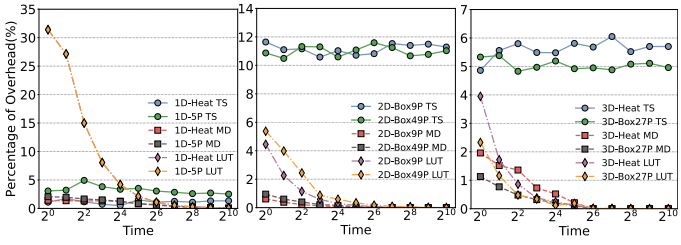


Figure 8: The percentages of overhead across various stencils. TS means Transformation, MD means Metadata and LUT means Lookup Table.

4.2 Preprocessing Overhead Analysis

Figure 8 illustrates the overhead associated with metadata, lookup table, and transformation across different stencils.

Overall overhead is minimal and quickly amortized. For 1D stencils, a brief spike is observed (1D-5P: LUT peaks near 30%) but rapidly diminishes. Higher-dimensional stencils, such as Box-3D27P, show consistently low and stable overhead due to favorable reuse patterns.

2D stencils (Box-2D9P, Box-2D49P) exhibit a slightly higher sustained overhead, occasionally exceeding 10%. Compared to 1D stencils with simpler memory access and 3D stencils with higher data reuse in tensor cores, 2D stencils have a less optimal balance between sparsity transformation and computation-to-memory ratio, leading to relatively higher preprocessing costs.

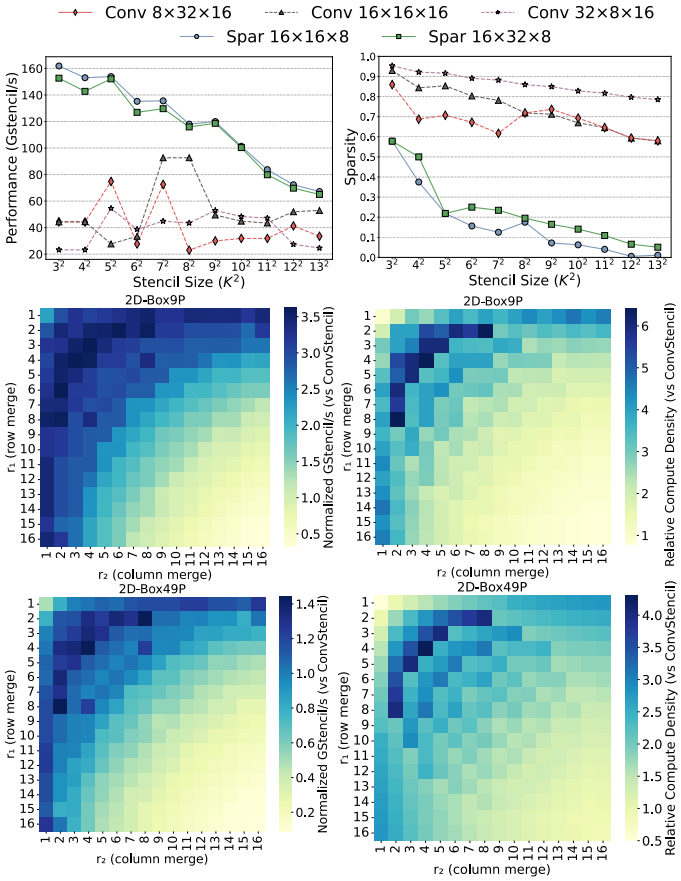


Figure 9: Performance, sparsity, and compute density across stencil sizes and layout configurations. Top: Throughput (left) and sparsity ratio (right) across stencil sizes on different TCU fragments. Bottom: Performance and compute density over (r_1, r_2) for two representative 2D stencil kernels.

4.3 State-of-the-art Comparison

Figure 6 compares SparStencil against state-of-the-art baselines. SparStencil consistently outperforms all reference methods, achieving 2.89x–60.35x speedup over cuDNN, which lacks Tensor Core

support for stencil patterns and underperforms on one-channel convolutions. In contrast, AMOS falls short due to inefficient stencil-to-TCU mapping.

Against stencil-specialized baselines like DRStencil and ConvStencil, SparStencil retains a clear advantage. On Star-2D13P and Box-2D49P, it achieves up to 38.89% higher performance, attributed to reduced memory traffic via layout search and superior compute throughput from sparse TCUs. This synergy of memory and compute underpins SparStencil's overall performance lead.

4.4 Performance Breakdown

Figure 7 illustrates the incremental performance gains from each optimization stage using Box-2D49P as a representative case. Applying Layout Morphing on dense TCUs yields a 1.58x speedup over the CUDA baseline, attributed to the superior matrix multiplication capabilities of dense TCUs. Subsequently, employing PIT to harness sparse TCUs results in an additional 1.22x speedup, as sparse TCUs deliver double the computational power of dense TCUs. Notably, for problem sizes 256 and 768, PIT resulted in 0.79x and 0.90x performance, respectively, as the memory overhead of PIT exceeds the compute efficiency gains from sparse TCUs at limited scales. Finally, the application of further performance-boosting optimizations achieves an additional 1.24x improvement. With this, we have completed all optimizations in SparStencil.

4.5 Adaptivity and Sparsity Across Varying Configurations

To better evaluate adaptivity, we omit the 3x temporal fusion used in Section 4.3, as it masks performance variation for small kernels. Here, we assess adaptivity and runtime sparsity across two dimensions: varying stencil sizes and diverse stencil patterns.

For varying stencil sizes, as illustrated in Figure 9, SparStencil consistently outperforms dense baselines across all stencil sizes and TCU fragments, achieving up to 4.1× speedup and maintaining sparsity below 60%. Heatmaps show that optimal compute density emerges in specific (r_1, r_2) regions, which are effectively discovered by our automated search. demonstrating the system's ability to adaptively align diverse stencil patterns with TCU-efficient layouts.

For diverse stencil patterns, as illustrated in Figure 10, across 79 stencil kernels spanning 9 application domains, SparStencil consistently achieves higher throughput, reaching up to 156.7 GStencil/s and outperforming cuDNN and ConvStencil by 6.3x and 3.1x on average, respectively. SparStencil maintains a stable performance, demonstrating robustness to stencil structural diversity. In terms of compute density, SparStencil achieves 17.92x and 4.46x over cuDNN and ConvStencil, respectively. Thanks to its adaptive layout search, SparStencil effectively reduces memory footprint and maintains stable performance even under extreme sparsity.

4.6 GPU Resource Utilization Comparison

Figure 11 shows hardware utilization across six metrics. SparStencil achieves higher SM utilization (74.5%) and occupancy (96.9%) than ConvStencil (18.3%, 61.3%) and cuDNN (59.4%, 88.5%), enabled by sparsity-aware layout. L1/TEX cache and memory throughput reach 64.5% and 64.1%, indicating effective on-chip reuse. In contrast, DRAM throughput (17.5%) and L2 cache activity (52.6%) are lower than cuDNN (43.5%, 61.6%). This reduction reflects SparStencil's layout-aware access patterns, which promote reuse in L1/shared memory, reducing dependence on L2 and minimizing global memory pressure.

4.7 Evaluating SparStencil Beyond FP16: Performance at FP64

Given that current sparse TCUs lack direct support for FP64 computation, our primary evaluation mainly focuses on FP16 precision. To further evaluate its broader applicability, we extend our analysis to FP64-enabled dense TCUs, benchmarking against state-of-the-art approaches as shown in Table 3.

Despite the absence of hardware-level sparsity acceleration, SparStencil consistently outperforms existing methods across all stencil configurations, achieving speedups ranging from 1.11x to 7.13x. This advantage stems from adaptive layout morphing and search strategy, which automatically restructures stencil matrices to optimize computational and memory throughput, ensuring high efficiency even without specialized sparse TCU acceleration.

These results validate two critical insights: (1) SparStencil remains effective and achieves performance leadership even on dense TCUs, demonstrating its robustness beyond FP16-restricted architectures. (2) Future sparse TCUs with FP64 support will further amplify SparStencil's benefits, as our sparse-aware optimization framework is inherently aligned with next-generation hardware trends.

5 Related Work

The optimization of stencil computations has been extensively studied on both CPUs [23, 46] and GPUs [27, 34, 37].

On CPUs, vectorization [17, 18, 23, 42], data reuse strategies [36, 40, 50], and tiling techniques, such as diamond [4, 7], time skewing [22, 43], rectangular [39], and tessellating tiling [46], are key methods for improving performance.

On GPUs, spatial tiling [13, 26, 50], temporal tiling [6, 14, 19, 29, 35, 41], unrolling [15], prefetching [38], and streaming [37] techniques effectively leverage GPU parallelism and memory hierarchies. Frameworks like Brick [50–52] exploit fine-grained data reuse, while DRStencil [44] applies fusion-partition optimization for stencil acceleration. Recent research also explores Tensor Cores, with TCStencil [24] applying them directly, ConvStencil [9] bridging convolution and stencil computations, FlashFFT [16] restructures FFT-based stencil computations on TCU and LoraStencil [49] reduces redundancy through low-rank decomposition of stencil. Furthermore, libraries like cuDNN [10, 30] optimize stencil-related

Table 3: Performance Comparison of SparStencil and SOTA methods under FP64 precision across different stencils (GFlops/s)

Method	Heat-2D	Box-2D9P	Star-2D13P	Box-2D49P
AMOS	10.16	10.23	10.51	10.59
cuDNN	64.33	64.57	17.05	17.15
DRStencil	55.46	57.63	50.16	20.28
ConvStencil	65.83	62.76	64.37	63.93
SparStencil	**72.49**	**73.25**	**71.34**	**67.28**

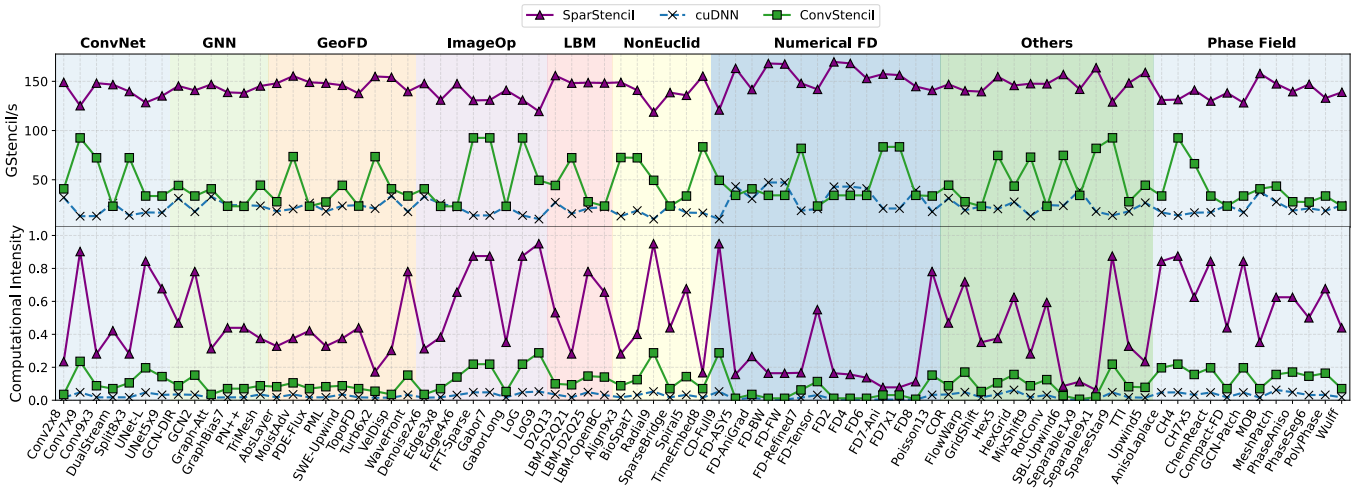


Figure 10: Performance and compute density breakdown across 79 real-world stencil kernels spanning 9 application domains. Top: end-to-end throughput (GStencil/s) of SparStencil, cuDNN, and ConvStencil; Bottom: corresponding compute intensity.

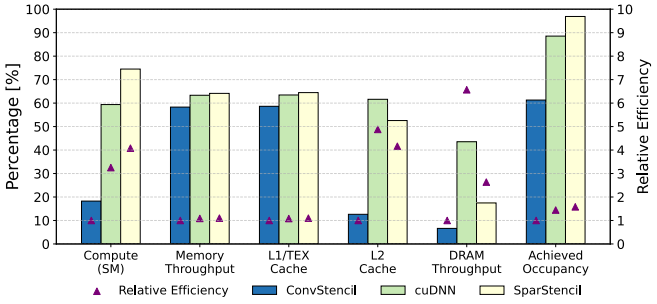


Figure 11: Hardware utilization comparison.

operations, and AMOS [53] maps depth-wise convolutions, equivalent to stencil operations, to hardware units, including Tensor Cores.

6 Conclusion

This paper presents SparStencil, a novel system leveraging sparse Tensor Cores for stencil computations via Permutation Invariant Transformation. It integrates Adaptive Layout Morphing, Structured Sparsity Conversion, and Automatic Kernel Generation to optimize stencil computations on sparse TCUs. Evaluations demonstrate its effectiveness, outperforming state-of-the-art approaches, with potential benefits for scientific and engineering applications.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful and constructive feedback. The work of Junshi Chen, and Hong An was supported in part by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDB0500102, and by Laoshan Laboratory under Grant No. LSKJ202300305.

References

[1] Yulong Ao, Chao Yang, Xinliang Wang, Wei Xue, Haohuan Fu, Fangfang Liu, Lin Gan, Ping Xu, and Wenjing Ma. 2017. 26 PFLOPS Stencil Computations for Atmospheric Modeling on Sunway TaihuLight. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 535–544. doi:10.1109/IPDPS.2017.9

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. The landscape of parallel computing research: A view from berkeley. (2006).

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A View of the Parallel Computing Landscape. *Commun. ACM* 52, 10 (oct 2009), 56–67. doi:10.1145/1562764.1562783

[4] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society, USA, 1–11. doi:10.1109/SC.2012.107

[5] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver D. Elbert, Rhea George, Jeremy McGibbon, Lukas Trümper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. 2022. Productive Performance Engineering for Weather and Climate Modeling with Python. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. doi:10.1109/SC41404.2022.00078

[6] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. 2017. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (May 2017), 1285–1298. doi:10.1109/TPDS.2016.2615094

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. doi:10.1145/1375581.1375595

[8] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 53, 81 pages. doi:10.1145/3295500.3356162

[9] Yuetao Chen, Kun Li, Yuhao Wang, Donglin Bai, Lei Wang, Lingxiao Ma, Liang Yuan, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. ConvStencil: Transform Stencil Computation to Matrix Multiplication on Tensor Cores. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 333–347. doi:10.1145/3627535.3638476

[10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[11] Alain Denzler, Rahul Bera, Nastaran Hajinazar, Gagandeep Singh, Geraldo F. Oliveira, Juan Gómez-Luna, and Onur Mutlu. 2023. Casper: Accelerating Stencil Computation using Near-cache Processing. arXiv:2112.14216 [cs.AR] https://arxiv.org/abs/2112.14216

[12] Jack Edmonds. 1965. Paths, Trees, and Flowers. *Canadian Journal of Mathematics* 17 (1965), 449–467. doi:10.4153/CJM-1965-045-4

[13] Thomas L. Falch and Anne C. Elster. 2014. Register Caching for Stencil Computations on GPUs. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 479–486. doi:10.1109/SYNASC.2014.70

[14] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (Houston, Texas, USA) *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 24–31. doi:10.1145/2458523.2458526

[15] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation. *ACM Trans. Archit. Code Optim.* 18, 4, Article 51 (sep 2021), 23 pages. doi:10.1145/3469030

[16] Haozhi Han, Kun Li, Wei Cui, Donglin Bai, Yiwei Zhang, Liang Yuan, Yifeng Chen, Yunquan Zhang, Ting Cao, and Mao Yang. 2025. FlashFFTStencil: Bridging Fast Fourier Transforms to Memory-Efficient Stencil Computations on Tensor Core Units. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 355–368. doi:10.1145/3710848.3710897

[17] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *Compiler Construction*, Jens Knoop (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 225–245.

[18] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A Stencil Compiler for Short-Vector SIMD Architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/2464996.2467268

[19] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 311–320. doi:10.1145/2304576.2304619

[20] H.T. Huynh, Z.J. Wang, and P.E. Vincent. 2014. High-order methods for computational fluid dynamics: A brief review of compact differential formulations on unstructured grids. *Computers & Fluids* 98 (2014), 209–220. doi:10.1016/j.compfluid.2013.12.007 12th USNCCM mini-symposium of High-Order Methods for Computational Fluid Dynamics - A special issue dedicated to the 80th birthday of Professor Antony Jameson.

[21] Mathias Jacquelin, Mauricio Araya–Polo, and Jie Meng. 2022. Scalable Distributed High-Order Stencil Computations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. doi:10.1109/SC41404.2022.00035

[22] Guohua Jin, J. Mellor-Crummey, and R. Fowler. 2001. Increasing Temporal Locality with Skewing and Recursive Blocking. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. 57–57. doi:10.1109/SC.2001.10041

[23] Kun Li, Liang Yuan, Yunquan Zhang, and Yue Yue. 2021. Reducing Redundancy in Data Organization and Arithmetic Calculation for Stencil Computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 84, 15 pages. doi:10.1145/3458817.3476154

[24] Xiaoyan Liu, Yi Liu, Hailong Yang, Jianjin Liao, Mingzhen Li, Zhongzhi Luan, and Depei Qian. 2022. Toward accelerated stencil computation by adapting tensor core unit on GPU. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–12.

[25] David J. Lusher, Satya P. Jammy, and Neil D. Sandham. 2021. OpenSBLI: Automated code-generation for heterogeneous computing architectures applied to compressible fluid dynamics on structured grids. *Computer Physics Communications* 267 (2021), 108063. doi:10.1016/j.cpc.2021.108063

[26] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st international workshop on high-performance stencil computations, Vienna*. Citeseer, 89–95.

[27] Naoya Maruyama, Kento Sato, Tatsuo Nomura, and Satoshi Matsuoka. 2011. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2063384.2063398

[28] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 199–211.

[29] Jiayuan Meng and Kevin Skadron. 2009. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 256–265. doi:10.1145/1542275.1542313

[30] Nvidia. 2023. cuDNN. https://developer.nvidia.com/cudnn, Last accessed on 2023-7-24.

[31] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Datasheet*. https://www.nvidia.com/en-us/data-center/a100/ Accessed: 2024-11-21.

[32] NVIDIA Corporation. 2024. *Parallel Thread Execution ISA Version 8.5*. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html Accessed: 2024-11-21.

[33] Jeff Pool, Abhishek Sawarkar, and Jay Rodge. 2021. Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/ Accessed: 2024-11-22.

[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[35] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. SDSLc: A Multi-Target Domain-Specific Compiler for Stencil Computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing* (Austin, Texas) *(WOLFHPC '15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages. doi:10.1145/2830018.2830025

[36] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 590–602. doi:10.1109/SC.2018.00049

[37] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2018. Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations. *Proc. IEEE* 106, 11 (2018), 1902–1920. doi:10.1109/JPROC.2018.2862896

[38] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2019. On Optimizing Complex Stencils on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 641–652. doi:10.1109/IPDPS.2019.00073

[39] G. Rivera and Chau-Wen Tseng. 2000. Tiling Optimizations for 3D Scientific Computations. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. 32–32. doi:10.1109/SC.2000.10015

[40] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A Framework for Enhancing Data Reuse via Associative Reordering. *SIGPLAN Not.* 49, 6 (jun 2014), 65–76. doi:10.1145/2666356.2594342

[41] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (jan 2013), 23 pages. doi:10.1145/2400682.2400713

[42] Luhan Wang, Haipeng Jia, Lei Xu, Cunyang Wei, Kun Li, Xianmeng Jiang, and Yunquan Zhang. 2024. VNEC: A Vectorized Non-Empty Column Format for SpMV on CPUs. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 14–25. doi:10.1109/IPDPS57955.2024.00011

[43] David Wonnacott. 2002. Achieving Scalable Locality with Time Skewing. *Int. J. Parallel Program.* 30, 3 (jun 2002), 181–221. doi:10.1023/A:1015460304860

[44] Xin You, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian. 2021. DRStencil: Exploiting Data Reuse within Low-order Stencil on GPU. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 63–70.

[45] Liang Yuan, Shan Huang, Yunquan Zhang, and Hang Cao. 2019. Tessellating Star Stencils. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 43, 10 pages. doi:10.1145/3337821.3337835

[46] Liang Yuan, Yunquan Zhang, Peng Guo, and Shan Huang. 2017. Tessellating Stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 49, 13 pages. doi:10.1145/3126908.3126920

[47] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. 2023. PERKS: a Locality-Optimized Execution Model for Iterative Memory-bound GPU Applications. In *Proceedings of the 37th International Conference on Supercomputing*. 167–179.

[48] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. 2023. Revisiting Temporal Blocking Stencil Optimizations. In *Proceedings of the 37th International Conference on Supercomputing*. 251–263.

[49] Yiwei Zhang, Kun Li, Liang Yuan, Jiawen Cheng, Yunquan Zhang, Ting Cao, and Mao Yang. 2024. LoRAStencil: Low-Rank Adaptation of Stencil Computation on Tensor Cores . In *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. IEEE Computer Society, Los Alamitos, CA, USA, 839–855. doi:10.1109/SC41406.2024.00059

[50] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. 2019. Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 52, 44 pages. doi:10.1145/3295500.3356210

[51] Tuowen Zhao, Mary Hall, Hans Johansen, and Samuel Williams. 2021. Improving Communication by Optimizing On-Node Data Movement with Data Layout. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 304–317. doi:10.1145/3437801.3441598

[52] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. 2018. Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 59–70. doi:10.1109/P3HPC.2018.00009

[53] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 874–887.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 This paper introduces SparStencil, the first system to retarget sparse Tensor Cores—originally designed for AI workloads—to accelerate stencil computations, a fundamental operator in scientific computing. By bridging the structural gap between scientific workloads and AI-native hardware, SparStencil opens a new paradigm for harnessing specialized accelerators in scientific applications.

C_2 We present an innovative Adaptive Layout Morphing technique, which reformulates diverse stencil patterns into unified matrix representations, exposing staircase-aligned sparsity. This transformation systematically reduces redundant data accesses and enhances hardware alignment, optimizing computational throughput.

C_3 We introduce a column-wise conflict graph-based approach for Structured Sparsity Conversion. This method guarantees optimal 2:4 compatibility with sparse Tensor Cores by leveraging a hierarchical two-level matching algorithm that resolves alignment conflicts while minimizing zero-padding and maximizing utilization.

C_4 Our system incorporates an advanced compilation framework that automatically explores the best computational layout, performs memory mapping, and generates high-performance code for sparse Tensor Cores. This automated process delivers expert-level optimization without requiring manual tuning, making it accessible and efficient for diverse stencil applications.

A.2 Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.15279052

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 6,7
	C_2,C_3	Figure 8,9
	C_4	Figure 10,11

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

The computational artifact A_1 is a stencil computing system that implements the core ideas introduced in this paper. It demonstrates how sparse Tensor Cores can be effectively retargeted for scientific stencil computations, leveraging AI-native hardware to achieve substantial speedups. The artifact includes all components of the SparStencil pipeline—from layout transformation to sparse format conversion and automatic kernel generation—providing a practical realization of contributions C_1, C_2, C_3, and C_4. It further enables quantitative evaluation across a wide range of real-world stencil workloads.

Expected Results

This artifact demonstrates that SparStencil enables stencil computations to effectively exploit sparse Tensor Cores. The experimental results confirm that (1) stencil workloads benefit significantly from adaptive layout and sparsity-aware compilation (validating C_1, C_2), (2) our graph-theoretic conversion preserves efficiency across a range of patterns (C_3), and (3) our compiler backend consistently delivers expert-level performance without manual tuning (C_4). Benchmarks across 79 stencil kernels substantiate these claims with speedups up to 7.1x and strong hardware utilization.

Expected Reproduction Time (in Minutes)

The expected computational time of kernel configuration in Table 2 within this artifact on NVIDIA A100 is anticipated to be under 1 minute. However, reproducing Figure 9 and Figure 10 may require 1–2 hours, as they involve automated layout searches across a large number of diverse stencil kernels to identify optimal configurations.

Artifact Setup (incl. Inputs)

Hardware.

- **CPU:** AMD EPYC 7V13 processor (or equivalent)
- **GPU:** NVIDIA A100 Tensor Core GPU (80GB, supports sparse TCU)
- **System:** Any Linux system supporting CUDA 12.4+ and GCC 11.4+

Software.

- **CUDA Toolkit:** 12.4
- **GCC:** 11.4.0

Datasets / Inputs. The datasets are provided within the computational artifact A_1

Installation and Deployment. Our artifact is available on Zenodo: https://doi.org/10.5281/zenodo.15279052. And it can be compiled using:

```
mkdir -p build
cd build
cmake ..
make
```

Artifact Execution

The following command serves to execute the compiled

```
sparstencil shape input_size time_size
```

where:

- `shape` specifies the stencil pattern and radius. Valid options include:
  - **1D:** `1d1r`, `1d2r` (corresponding to Heat1D and 1D5P in the paper)
  - **2D:** `star2d1r`, `box2d1r`, `star2d3r`, `box2d3r` (corresponding to Heat2D, Box2D9P, Star2D13P, Box2D49P)
  - **3D:** `star3d1r`, `box3d1r` (corresponding to Heat3D, Box3D27P)

- `input_size` defines the spatial dimensions of the problem. The number of values expected equals the stencil dimensionality (e.g., 2 values for 2D, 3 values for 3D).
- `time_size` denotes the number of iterations (timesteps).

Artifact Analysis (incl. Outputs)

Upon executing the above command, SparStencil will generate the following outputs, which include computation time and speed (measured in GStencil/s).

```
./sparstencil_ box2d1r 10240 10240 10240
SparStencil(2D box_2d1r):
Time = xxx[ms]
GStencil/s = xxx
```

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

Our artifact is available on https://doi.org/10.5281/zenodo.15279052. And it can be compiled using:

```
mkdir -p build
cd build
cmake ..
make
```

Artifact Execution

The CMake output will be as follows:

```
-- The CXX compiler identification is GNU
11.4.0
-- The CUDA compiler identification is NVIDIA
12.3.107
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c
++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting CUDA compiler ABI info
-- Detecting CUDA compiler ABI info - done
-- Check for working CUDA compiler: /usr/local
/cuda/bin/nvcc - skipped
-- Detecting CUDA compile features
-- Detecting CUDA compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/
your_path/SparStencil/build
```

Artifact Execution

Following compilation, an executable file will be generated in the build/ directory:

- `SparStencil`

`SparStencil` can be executed using the following input format:

```
sparstencil shape input_size time_size
```

We also provide a convenient script to automatically test all stencil shapes:

- test_all.sh

This script compiles the project and executes benchmark tests for all supported stencil patterns with default configurations.

Artifact Analysis (incl. Outputs)

Upon executing the above script, SparStencil will generate the following outputs, which include computation time and speed (measured in GStencil/s). We use Box2D9P as an example.

```
[SparStencil Box_9P] Time = 6007[ms]
321.745944
```