Jigsaw: Toward Conflict-free Vectorized Stencil Computation by Tessellating Swizzled Registers

Yiwei Zhang*†
University of Chinese Academy of Sciences
Microsoft Research
Beijing, China

Kun Li†‡
Microsoft Research
Beijing, China

Liang Yuan‡
Chinese Academy of Sciences
Beijing, China

Haozhi Han*
Peking University
Microsoft Research
Beijing, China

Yunquan Zhang
Chinese Academy of Sciences
Beijing, China

Ting Cao
Microsoft Research
Beijing, China

Mao Yang
Microsoft Research
Beijing, China

Abstract

Stencil computation plays a pivotal role in numerous scientific and engineering applications. Previous studies have extensively investigated vectorization techniques to enhance in-core parallelism; however, the performance bottleneck caused by data alignment conflicts (DAC) has not been effectively resolved in all dimensions. This paper proposes *Jigsaw*, a conflict-free vectorization method to reduce DAC across all dimensions by tessellating swizzled finest-grained lanes. Jigsaw comprises three key components: Lane-based Butterfly Vectorization, SVD-based Dimension Flattening, and Iteration-based Temporal Merging. These components effectively address DAC across spatial and temporal dimensions. Experimental results on different machines demonstrate that Jigsaw could achieve a significant improvement compared to the state-of-the-art techniques, with an average speedup of 2.31x on various stencil kernels.

CCS Concepts: • **Computing methodologies** → **Vector / streaming algorithms**; • **Theory of computation** → **Vector / streaming algorithms**.

*Work done during an internship at Microsoft Research, with Project Lead (kunli@microsoft.com).
†Both authors contributed equally to this research.
‡Corresponding author.

This work is licensed under a Creative Commons Attribution 4.0 International License.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710886

Keywords: Stencil Computation, High Performance Computing, Vectorization, Data Alignment Conflict

1 Introduction

Ubiquitous in scientific or industrial computing, stencil computation is identified as one of the principal templates in the high performance computing community [4, 5]. The essence of stencil computation lies in a pre-defined pattern that iteratively updates given points using neighboring grid points [51]. The naive computation of a d-dimensional stencil is accomplished by $d+1$ nested loops, with the outermost updating along the time dimension and the inner iterates over each grid point. Consequently, stencil computation suffers from poor data reuse and low computational intensity, notoriously known as a memory-bound kernel [11, 32, 61].

Various optimization techniques for stencil have been exhaustively studied in order to improve performance, among which vectorization has been demonstrated as an effective approach [22, 30, 35, 38, 41, 48, 69]. Leveraging the SIMD facilities in modern CPU architectures, vectorization seeks to boost in-core throughput by exploiting data-level parallelism. Although vectorization is prevalent and promises performance improvements, it is still critically bottlenecked by an accompanying problem, *data alignment conflict*.

Data alignment conflict (DAC) is the main performance-limiting factor caused by vectorization on stencil computation inherently. In the iteration space, it manifests as vector-data conflict in the innermost loop and vector-dimension conflict in the outer loops. In the innermost loop, i.e. unit-stride update direction, since the data elements are stored contiguously in memory, the neighbors for each element are loaded into different positions within the same register. However, stencil computation requires remapping adjacent elements to the same position in different registers, shaping a

conflict between vector operations and data layout. When extended to multi-dimensional stencils, this conflict propagates to other dimensions (outer loops), introducing exponentially growing additional vector-data conflicts, which we refer to as vector-dimension conflict.

Significant efforts have been devoted to alleviating this issue [24, 61], by reducing the volume of loading data and/or shuffle instructions. Recently, one milestone approach to address DAC is *Folding* [37], which endeavors to exploit data reuse by optimizing the vectorization order. This method stands as a pinnacle in the realm of stencil vectorization. However, its in-register transposition introduces a significant number of non-computational shuffle instructions, reducing computational intensity. Moreover, akin to other vectorization attempts, it focuses on high-level optimization of stencil algorithms while neglecting the underlying architecture, thus failing to fully exploit the hardware's potential.

In this paper, we propose a conflict-free vectorization method for stencil computation, ***Jigsaw***, to efficiently reduce DAC across all dimensions by tessellating swizzled registers with the finest-grained lanes.

The design of Jigsaw is based on two key observations, which first delve into the deeper roots of DAC in terms of vector register architecture: 1) Existing works primarily follow a top-down methodology. They focus solely on optimizing stencil at the algorithmic level without delving into the underlying vector architecture, which leads to an underutilization of the hardware's full potential. 2) Vector registers are composed of the finest-grained operational units called lanes. Cross-lane instructions are significantly more expensive than in-lane instructions and these register reorganization operations constitute a major non-computational bottleneck in existing work.

Guided by upon observations, the key insight of Jigsaw is to adopt a bottom-up methodology, where the design of the stencil vectorization algorithm is informed by the underlying architecture. By employing tessellating swizzled manipulations of the finest-grained lanes within vector registers, Jigsaw provides a general and flexible solution to the DAC. Consequently, this approach enables conflict-reduced vectorized computations across single-dimensional, multi-dimensional, and temporal dimensions.

Jigsaw incorporates three key techniques: Lane-based Butterfly Vectorization for single dimension, SVD-based Dimension Flattening for multiple dimensions, and Iteration-based Temporal Merging for temporal dimension.

Lane-based Butterfly Vectorization (LBV) mitigates vector-data conflicts in the innermost spatial loop by meticulously manipulating finest-grained lanes. Capturing the hardware characteristics of vector registers, LBV alleviates vector-data conflicts by moving data lane-to-lane, which minimizes the expensive cross-lane overhead to the theoretical lower bound. Moreover, LBV overlaps data-reordering and arithmetic operations carefully to occupy different vector functional units on the CPU. Unlike previous work addressing DAC before or after stencil computation, this design further decreases pipeline bubbles caused by data layout transformation in computation.

SVD-based Dimension Flattening (SDF) addresses vector-dimension conflicts in the outer spatial loops by mathematically decomposing the coefficient matrix. By decomposing the stencil coefficient matrix into an overlay of several rank-1 matrices, SDF reorganizes the computation order at the vector register level, transforming 2D stencil into 1D stencils. By leveraging the consistency of lane dependency across multiple loops, SDF reduces the redundant vector reorganization instructions in high-dimensional stencils. This method minimizes unnecessary inter/intra-register data shuffling and alleviates register spilling, reducing data preparation time and computation-agnostic pipeline stalls.

Iteration-based Temporal Merging (ITM) further reduces data alignment conflict in the outermost temporal loop by merging computations along the time dimension. As on-chip register space is limited and hard to be employed for temporal data reuse, the updates are swept from registers to cache instantly as a usual practice. Here, we design a novel temporal compression strategy to explore multi-step stencil computations within a single iteration on registers. It identifies the temporal dependencies and carefully compresses the register footprints to eliminate recurring DACs in multi-step iterations, thereby efficiently reducing the data transfer volume between registers and cache.

To the best of our knowledge, Jigsaw is the first stencil vectorization scheme that captures hardware characteristics and employs a bottom-up design to mitigate DAC. This method tessellates vector registers in both spatial and temporal dimensions to achieve conflict reduction across all dimensions. Furthermore, benefiting from Jigsaw's flexible design that delves deeply into the underlying architecture, it is orthogonal to other high-level optimization techniques, such as blocking and computation reordering.

We evaluate Jigsaw on both Intel and AMD architectures against classical vectorization algorithms (Multiple Loads [52], Multiple Permutations [9, 61]), highly optimized domain-specific languages (DSLs) (SDSL [24], Pluto [6, 8]), and state-of-the-art optimization work (Folding [60], Tessellation [37]), experimental results demonstrate the effectiveness of Jigsaw.

Our contributions are outlined as follows:

- We propose Jigsaw, a novel stencil vectorization method that delves into the underlying architecture of vector registers to reduce DAC across all dimensions.
- LBV, SDF and ITM achieve comprehensive reductions in DAC across single-dimensional, multi-dimensional, and temporal dimensions, respectively.
- We implement Jigsaw and experimental results demonstrate the superior efficiency of Jigsaw compared to various state-of-the-art methods.

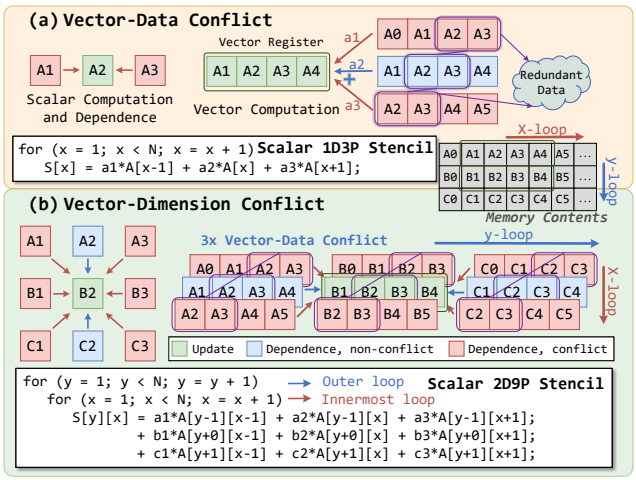

Figure 1. Data Alignment Conflicts (DAC) in the spatial dimension of stencil vectorization. Left side depicted scalar computations for 1D3P and 2D9P stencils, while right illustrated vector-data conflicts introduced in the innermost dimension and vector-dimension conflicts introduced in the outer spatial dimensions during vectorization.

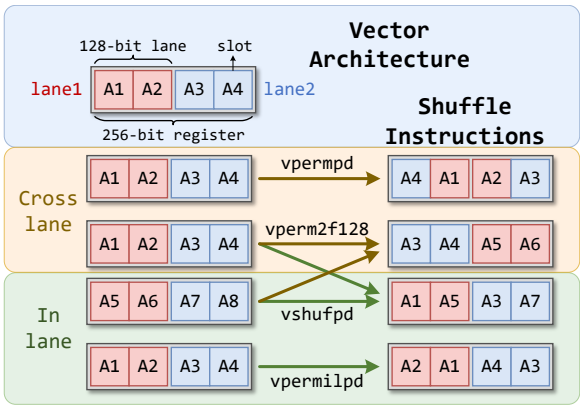

Figure 2. Vector Architecture and Shuffle Instructions. Elements of the same color within a vector indicate that they originally reside in the same lane.

2 Background

2.1 Data Alignment Conflict

Stencil computation performs iterative updates on grid points within multidimensional inputs according to a predefined computational pattern. It is commonly denoted as nDkP to indicate the dimensions and number of points involved. Figure 1 illustrates the stencil computation patterns and data alignment conflict (DAC) of 1D3P and 2D9P. DAC is a critical bottleneck in stencil vectorized computation, primarily caused by data dependencies. We elucidate this fundamental issue via two straightforward examples.

Figure 1(a) illustrates the vector-data conflict issue encountered during the vectorized computation of a 1D3P stencil. In scalar computation, elements within registers can be shift-reused, ensuring that each point in the iteration space is loaded from memory only once. However, in vectorized computation, elements within vector registers are not reusable in the subsequent iteration, leading to a significant increase in memory accesses. For instance, in Figure 1(a), the three vectors are discarded after computing `(A1,A2,A3,A4)`, and cannot be reused for computing `(A5,A6,A7,A8)`.

Figure 1(b) illustrates the extension of vector-data conflicts to vector-dimension conflicts in a 2D9P stencil. It can be observed that the vector-data conflicts present in the unit-stride update dimension of the spatial domain (i.e., the innermost loop x) are extended to the outer loop y, resulting in a threefold increase in conflicts compared to the 1D3P case. This escalation occurs because vector operations are confined to a single dimension; hence, when multi-dimensional dependencies are introduced, they lead to multiple instances of additional vector-data conflicts.

The crux of data alignment lies in the requirement for vectorized operations to aggregate adjacent elements into a single vector register. However, the unique nature of stencil computation, where each element's iteration depends on its neighboring elements, introduces a need for data dependencies across different positions within the register. This results in additional data movement instructions or redundant memory access to satisfy inter-vector dependencies, thereby diminishing the efficiency of vectorized computations.

Significant efforts have been devoted to alleviating vector-data conflict. *Multiple Loads* adopts a straightforward implementation by staggered loadings from memory [52]. It achieves an efficient computing pipeline without any shuffle bubbles, while the data transfer volume is multiplying at a dizzying rate. Moreover, unaligned data access introduced by staggered loading degrades the performance considerably. On the contrary, *Multiple Permutations* loads each element into the register only once and assembles the required vectors via inter/intra-register shuffle instructions [9, 61]. Compared with the previous method, it reduces memory bandwidth usage and takes advantage of the rich set of data-reordering instructions on the CPU. However, it produces massive non-compute bubbles in the pipeline, and the limited data shuffle units exacerbate the pressure of data-reordering traffic inside the CPU.

2.2 Architecture of Vector Registers

The vector register is a fundamental unit of SIMD (Single Instruction Multiple Data) architectures for instruction operations and data storage in vector computations. In modern CPUs, all vector registers (SSE-style 128-bit, AVX/AVX2-style 256-bit, or Intel AVX-512 style 512-bit) can be divided into 128-bit groups known as lanes [25]. For instance, in the AVX2 instruction set architecture, a vector register YMM

is constructed by linking two 128-bit lanes to form a cohesive unit, and each lane is an XMM vector register, as illustrated in Figure 2. To enhance the vectorization opportunities, register-based gather-scatter instructions, such as shuffle instructions are indispensable. These instructions enable the movement and reordering of data between registers, making them applicable to more vectorized scenarios. Due to the lane-based architecture design, cross-lane instructions (`vpermpd`, `vperm2f128`) incur additional terms of instruction execution and data communication compared to in-lane instructions (`vshufpd`, `vpermilpd`). Consequently, this results in significant latency and throughput penalties [20], as shown in Table 1.

3 Jigsaw

3.1 Lane-based Butterfly Vectorization

In this subsection, we introduce a novel stencil vectorization method, *Lane-based Butterfly Vectorization*, aimed at minimizing the shuffle overhead induced by vector-data conflicts in the innermost dimension.

The development of LBV is predicated upon two key observations: 1) Stencil vectorization introduces a substantial number of vector shuffle instructions to construct dependent vectors. Minimizing these non-computational register data movement costs is crucial for boosting performance. 2) The efficiency of shuffle instructions is contingent upon the vector architecture, with vectors being composed of finer-grained lanes. Consequently, cross-lane instructions are much more expensive than in-lane instructions. Previous vectorization approaches have focused on the vector level as the minimal granularity, inadvertently introducing numerous expensive cross-lane instructions to construct dependent vectors. These cross-lane instructions can be reduced through tessellating swizzled manipulations of lanes.

Drawing from the above observations, we propose LBV by conducting butterfly vectorization at the finest granularity, the lane level. LBV exploits the architectural features of vector registers, employing cost-effective in-lane instructions to move adjacent data into neighboring registers, thereby satisfying half of the data dependencies necessitated for two vectors simultaneously. Additionally, by temporarily transforming the data layout within registers during computation, LBV facilitates the overlapping execution of dependency construction and data computation.

Table 1. Comparison of latency and throughput for Cross-lane and In-lane instructions in Alder/Ice Lake architectures.

Type	Cross-lane		In-lane	
Instruction	vpermpd	vperm2f128	vshufpd	vpermilpd
Latency	3	3	1	1
Throughput (CPI)	1	1	0.5	1

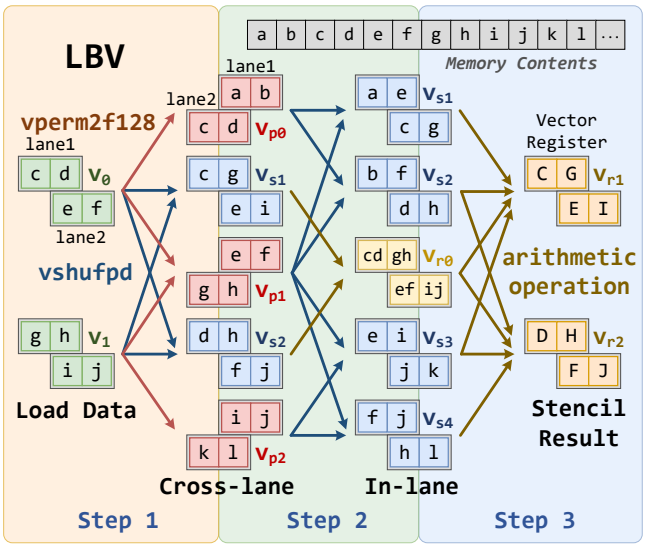

Figure 3. Lane-based Butterfly Vectorization of Jigsaw. The red, blue, and yellow arrows represent cross-lane, in-lane, and arithmetic instructions, respectively. The blocks corresponding to these colors denote the vector registers obtained after executing the respective instructions.

LBV Process. Algorithm 1 and Figure 3 illustrate the detailed computational process of LBV using 1D5P stencil as an example. In Figure 3, we depict the vector register as Tetris blocks to illustrate their underlying structure, with the upper and lower blocks representing `lane1` and `lane2`, respectively. In Algorithm 1, the outermost loop (line 2) iteratively sweeps along the time dimension. Lines 3-4 prefetch two vectors $\mathbf{v}_0$ and $\mathbf{v}_{p0}$ for boundary processing and subsequent pipelined computation. The innermost loop employs the LBV method to update grid points with a stride of two vector lengths (vl=4), which encompasses three steps.

Step 1 corresponds to lines 6-10 of Algorithm 1. Initially, vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are loaded to update the elements in $\mathbf{v}_0$ and $\mathbf{v}_1$ for the next step, with the loaded and updated elements shown as green and orange blocks in Figure 3, respectively. The only cross-lane instruction used in LBV appears in lines 9-10, implemented via the `vperm2f128` instruction to concatenate lanes from two vectors, i.e., `(lane1|lane2)+(lane3|lane4)`$\rightarrow$`(lane2|lane3)`. Concurrently, the in-lane instruction `vshufpd`, which has lower latency, is used to exchange data within the corresponding lanes of the two vectors, i.e., $(c,d\,|\,e,f)+(g,h\,|\,i,j)\rightarrow(c,g\,|\,e,i)+(d,h\,|\,f,j)$. The elements generated by cross-lane and in-lane operations are represented by red and blue blocks, respectively.

Step 2 corresponds to lines 11-13. After obtaining $\mathbf{v}_{p1}$ and $\mathbf{v}_{p2}$, in-lane shuffles are subsequently performed to construct additional vectors that satisfy dependencies on multiple neighboring points. Simultaneously, arithmetic operations can be conducted on $\mathbf{v}_{s1}$ and $\mathbf{v}_{s2}$, which were derived from

Algorithm 1 Lane-based Butterfly Vectorization for 1D5P Stencil. $T\%2 = 0, NX\%8 = 0$

```
1: function LBV( )
2:     for t ← 1 to T do
3:         v_0 ← (a_2^t, a_3^t, a_4^t, a_5^t)
4:         v_p0 ← (a_0^t, a_1^t, a_2^t, a_3^t)
5:         for x ← 2 to NX by 8 do
6:             v_1 ← (a_{x+4}^t, a_{x+5}^t, a_{x+6}^t, a_{x+7}^t)
7:             v_2 ← (a_{x+8}^t, a_{x+9}^t, a_{x+10}^t, a_{x+11}^t)
8:             v_s1, v_s2 ← InLaneShuffle(v_0, v_1)
9:             v_p1 ← CrossLaneShuffle(v_0, v_1)
10:            v_p2 ← CrossLaneShuffle(v_1, v_2)
11:            v_r0 ← ArithmeticOp(v_s1, v_s2)
12:            v_s1, v_s2 ← InLaneShuffle(v_p0, v_p1)
13:            v_s3, v_s4 ← InLaneShuffle(v_p1, v_p2)
14:            v_r1, v_r2 ← ArithmeticOp(v_s1, v_s2, v_r0, v_s3, v_s4)
15:            v_0, v_p0 ← v_2, v_p2
16:            v_r1, v_r2 ← InLaneShuffle(v_r1, v_r2)
17:            (a_x^{t+1}, a_{x+1}^{t+1}, a_{x+2}^{t+1}, a_{x+3}^{t+1}) ← v_r1
18:            (a_{x+4}^{t+1}, a_{x+5}^{t+1}, a_{x+6}^{t+1}, a_{x+7}^{t+1}) ← v_r2
19:        end for
20:    end for
21: end function
```

the previous in-lane shuffle, yielding partial results $(c+d, g+h \mid e+f, i+j)$ (depicted by yellow blocks in Figure 3). This strategy overlaps data movement with arithmetic computation, thereby reducing pipeline stalls.

Step 3 corresponds to lines 14-18, involves arithmetic computation and data storage. Once in-lane shuffles are completed in the second step, all dependency vectors required for the update are obtained. Performing arithmetic operations on these vectors yields the final stencil result. Following this, an in-lane shuffle is applied to $\mathbf{v}_{r1}$ and $\mathbf{v}_{r2}$, allowing the computed results to be written back to memory. By distributing the shuffles introduced by vector-data conflicts throughout the entire computation process via temporary data layout transformations in registers, we effectively reduce idle time waiting for data preparation during vector operations.

Instruction Efficiency Analysis. In LBV, the computation for each vector necessitates just one cross-lane instruction, reaching the theoretical lower bound. This arises from the inherent characteristics of stencil vectorization, where an element cannot reside exclusively within a single lane. Otherwise, all its neighbors and their neighbors would also have to reside within the same lane, which is infeasible. Consequently, with only one initial load, at least one cross-lane instruction is inevitably required.

Conversely, in Multiple Permutations, each neighbor dependency necessitates a separate cross-lane instruction, resulting in a linear increase in the number of cross-lane instructions as the stencil radius grows. This significantly impacts data preparation time as the radius increases. Multiple Loads encounters a similar problem. For example, from 1D3P to 1D5P, the number of vectors that must be loaded to compute a single vector increases from 3 to 5. Each load instruction (`vmovupd`) requires 7 clock cycles, which far exceeds the time needed for shuffle instructions, leading to severe pipeline inefficiencies.

Compared to the state-of-the-art Folding technique, which leverages matrix transposition for vectorized computation, LBV also reduces the number of cross-lane instructions by half. Additionally, Folding requires extensive cross-lane shuffling before each computation to transpose data within vector registers, whereas LBV overlaps shuffling with computation. This overlap significantly reduces pipeline stalls and alleviates concerns about register spilling. Furthermore, LBV's design, being inherently flexible and architecture-based, ensures excellent generality and scalability. It is not constrained by register length or specific application scenarios, making it a versatile solution.

3.2 SVD-based Dimension Flattening

After introducing the LBV computation for 1D stencil, the subsequent challenge is to address vector-dimension conflicts in multi-dimensional stencils. To this end, we propose *SVD-based Dimension Flattening* method to efficiently reduces conflicts across all spatial dimensions.

The fundamental idea behind SDF is to capitalize on the uniformity of innermost vectorized operations within the multi-layer spatial loops of stencil computations. By employing conflict-free vector gathering (Dimension Flattening) of dependencies in non-unit-stride dimensions, SDF eliminate vector-dimension conflicts in outer spatial dimensions, transforming 2D stencil computations into conflict-free 1D stencil computations. Algorithm 2 and Figure 4 illustrate the application of SDF and LBV in 2D stencil computation.

Dimension Flattening. The essence of vector-dimension conflicts lies in the diagonal dependencies of update points on cross-dimensional neighbors. By decomposing diagonal dependencies into update direction (innermost loop) and orthogonal direction (outer loop) dependencies, we can eliminate redundant shuffle instructions. This means first collecting conflict-free dependencies in the outer loop, followed by conflict dependency collection in the innermost loop.

We identify that dependencies in a single dimension can be encapsulated by a coefficient vector. When the coefficient matrix $C_{n\times n}$ (the matrix composed of weights corresponding to each point of the stencil) is of rank-1, it can be rank-decomposed into the outer product of two vectors (line 11), like Equation (1)

$$C = \boldsymbol{u} \otimes \boldsymbol{v}^{\mathrm{T}} \tag{1}$$

where $\boldsymbol{u}$ and $\boldsymbol{v}$ are the vertical and horizontal dependency vectors, respectively.

This rank-decomposition allows us to flatten the dependencies of a 2D stencil into a 1D stencil. The `Flattening` function in Algorithm 2 (line 1) represents the conflict-free

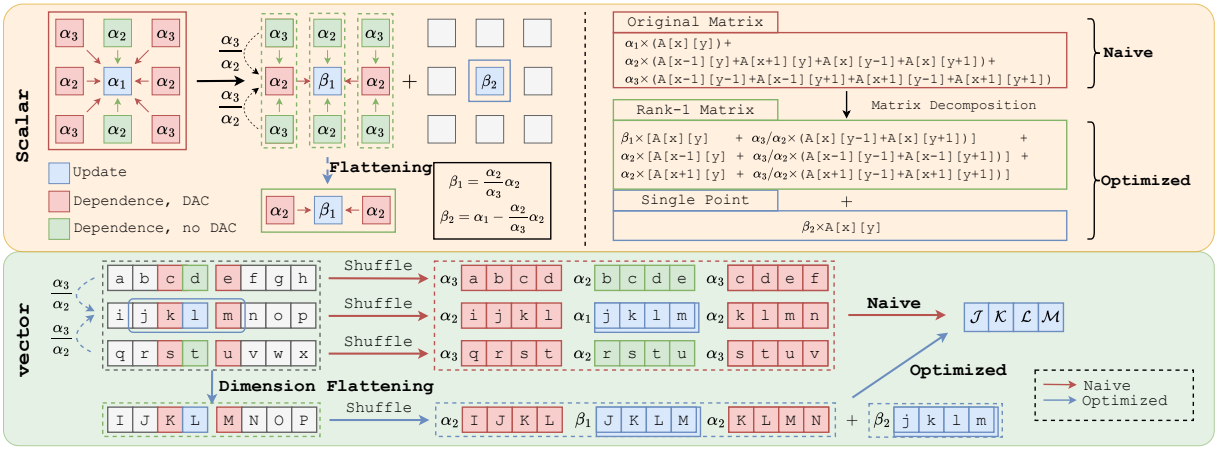

Figure 4. Scalar and vectorized illustration of SVD-based Dimension Flattening strategy for Box-2D9P stencil.

Algorithm 2 SVD-based Dimension Flattening and LBV.

```
1:  function Flattening(VS, u)
2:      for x ← 1 to n do
3:          for y ← 1 to n do
4:              v_x ← v_x + u_y × VS_yx
5:          end for
6:      end for
7:      return v_1, ..., v_n
8:  end function
9:  function Stencil( )
10:     C_1, ..., C_r ← SVDDecomposition(W_n×n) ▷ rank(W_n×n) = r
11:     (u_1, v_1), ..., (u_r, v_r) ← RankDecomposition(C_1, ..., C_r)
12:     for t ← 1 to T do
13:         for y ← 1 to NY do
14:             for x ← 2 to NX by 8 do
15:                 VS_2d ← VecLoad(A, y, x)
16:                 for i ← 1 to r do
17:                     VS_1d ← Flattening(VS_2d, u_i)
18:                     VS_r ← LBV(VS_1d, v_i)
19:                 end for
20:                 VecStore(A, y, x) ← VS_r
21:             end for
22:         end for
23:     end for
24: end function
```

vertical dependency collection for $n \times n$ vector registers, resulting in n vector registers that only have horizontal dependencies, thereby transforming the 2D stencil into a 1D stencil. Subsequently, the LBV method can be applied to compute the 1D stencil.

SVD Decomposition. However, for most stencils, their coefficient matrices are not naturally rank-1. To address this issue, we employ Singular Value Decomposition (SVD) to decompose the original coefficient matrix of any rank. Specifically, for a rank-r matrix $W_{n\times n}$, let $W = U\Sigma V^{\mathrm{T}}$ of the SVD of W, where U and V are orthogonal and $\Sigma = \mathrm{diag}(\sigma_1, ..., \sigma_n)$. We usually take $\sigma_1 \geq \sigma_2 \geq ...$, so $\sigma_i = 0$ for $i > r$. Define

$$\Sigma_i := \mathrm{diag}(0, ..., 0, \sigma_i, 0, ..., 0)$$

i.e., Σ_i has σ_i at i-th position and zeros everywhere else. Obviously, $\Sigma = \sum_i \Sigma_i$ and $\Sigma_i \neq 0$ if and only if $i \in \{1, ..., r\}$, so

$$W = U\Sigma V^{\mathrm{T}} = \sum_{i=1}^{r} U\Sigma_i V^{\mathrm{T}} = \sum_{i=1}^{r} C_i \tag{2}$$

then, we decompose W into a sum of r rank-1 matrix C_i (line 10). Subsequently, each C_i can be subjected to Dimension Flattening and LBV computation.

Coefficient Symmetry. Additionally, we observe that the coefficients often exhibit symmetry, meaning that neighbor points with the same Euclidean distance of their corresponding dependence directions share an identical coefficient [7, 13, 27, 37, 58]. This observation indicates that the original matrix has low-rank properties, resulting in only a few rank-1 matrices after matrix decomposition, which significantly reduces the subsequent computational workload.

For instance, in the case of the Box-2D9P stencil, its vectorized computation process is illustrated in Figure 4. Due to the symmetry of the coefficients, the original weight matrix can be decomposed into the sum of a rank-1 3×3 matrix along with a single point. This rank-1 coefficient matrix eliminates vector-dimension conflicts in the 2D space, thereby enabling direct dimension flattening and vector computation. By employing the SDF method, the vectorized computation of the 2D9P stencil is streamlined into an LBV computation for a 1D3P stencil and one multiply-and-accumulation operation. This method substantially minimizes redundant data movement and computational overhead.

Redundancy Reduction Analysis. By flattening dimensions to eliminate vector-dimension conflicts, we significantly reduce the redundant shuffle instructions required for high-dimensional stencil computations. For example, when applied to a 2D9P stencil, the SDF method reduces the total

Table 2. Analytical vector instructions for Jacobi Stencils (per vector)

Kernel	Star-1D5P				Box-2D9P				Box-3D27P				Heat-1D				Heat-2D				Heat-3D			
Operation [1]	L	S	C	I	L	S	C	I	L	S	C	I	L	S	C	C	L	S	C	I	L	S	C	I
Auto [2]	5	1	0	0	9	1	0	0	27	1	0	0	3	1	0	0	5	1	0	0	7	1	0	0
Reorg	1	1	3	3	3	1	6	6	9	1	18	18	1	1	2	2	3	1	2	2	5	1	2	2
Jigsaw	0.5	0.5	0.5	2	2.5	0.5	0.5	1	12.5	0.5	0.5	1	0.5	0.5	0.5	1.5	2.5	0.5	0.5	1	6.5	0.5	0.5	1

[1] For brevity, Load, Store, the Cross-Lane and In-Lane operations are abbreviated as L, S, C and I, respectively.
[2] Methods for Multiple Load, Data Reorganization and Jigsaw are also abbreviated accordingly (similarly hereinafter).

register shuffle instructions by 2/3, including 2 cross-lane instructions and 6 in-lane instructions. This significantly reduces the preparation necessitated before calculations in the innermost loop, effectively eliminating irrelevant pipeline bubbles and enhancing computational intensity.

Additionally, this method is a general-purpose computing technique that can be effortlessly applied to higher-order and higher-dimensional stencils in the same manner, and can result in better redundancy elimination optimization. For instance, for a Box-3D27P stencil, the SDF strategy can reduce 8/9 data shuffling work and make the box stencil achieve vectorization without redundant data movement.

3.3 Iteration-based Temporal Merging

LBV and SDF significantly reduce the data movement overhead caused by DAC in the spatial dimensions. However, redundant shuffle instructions and cache-register data transfers remain problematic in the temporal dimension. To address this issue, we propose *Iteration-based Temporal Merging* method, which aims to reduce DAC across all dimensions. By merging multiple iterative time steps, we facilitate intra-register data reuse in the temporal dimension. Figure 5 illustrates the ITM and subsequent SDF computations of Jigsaw, using the Star-2D5P stencil as an example.

As shown in Figure 5, for the 2D5P stencil, the update of an element directly depends on its adjacent red and green neighbors and indirectly on the white neighbors. If updates are iteratively performed to obtain the result after two steps, scalar computation would necessitate $5 \times 5 + 5 = 30$ memory accesses, whereas vectorized computation would require $5 \times 1 + 1 = 6$ cross-lane shuffle instructions. However, the dependent grid points necessitated by the update total only 13, indicating that the necessary memory access and vector dependencies are significantly fewer than the iterative access count and shuffle instructions.

Hence, we can unfold and merge the coefficients along the temporal dimension to directly accomplish two-step computations, achieving local fusion of iterations. Specifically, this entails squaring the old coefficients (α) to derive new coefficients (β, γ), which are then distributed to the 13 grid points required for the two-step computation. Given that the stencil extended by ITM introduces diagonal dependencies, we employ SDF to eliminate additional vector-dimension

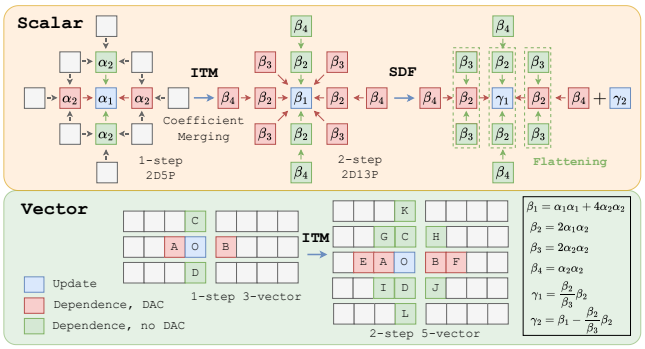

Figure 5. Iteration-based Temporal Merging of Jigsaw. After applying ITM to the 2D5P stencil, it transforms into a 2D13P stencil. Subsequently, SDF is utilized to eliminate vector-dimension conflicts within the 2D13P stencil.

conflicts. Additionally, it is noteworthy that the new coefficient matrix (the central 9 points) retains its symmetric properties, thus not incurring extra computational overhead.

Data-movement Reduction Analysis. Following the application of ITM, the data transfer volume between cache and registers in each step of stencil vector computation is reduced, alleviating bandwidth pressure. Specifically, we note that the number of registers required for 2D stencil vector computation is determined solely by the number of vertical elements and is unaffected by the number of horizontal elements, as horizontal elements can be reused through in-register shifts. Consequently, the 2D5P stencil necessitates $5/2 = 2.5$ vector registers per step after ITM, which is fewer than the 3 registers required for a single-step iteration. Moreover, the data transfer volume between registers in each computation step is significantly reduced, eliminating numerous non-computational shuffle instructions. Specifically, after ITM, each step requires only $1/2 = 0.5$ cross-lane shuffle instructions, while redundant in-lane instructions are eliminated, significantly enhancing computational density.

Additionally, ITM can perform multi-step fusion on 1D stencils to further reduce redundant data transfers. For instance, by applying a three-step fusion to a 1D3P stencil, we can reduce the number of vector load and store instructions to 1/3 of the original, as illustrated in Figure 6. Simultaneously, the cross-lane and in-lane instructions are reduced to

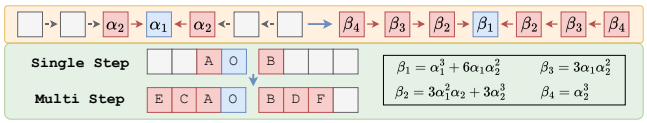

Figure 6. ITM for 1D3P stencil with 3-step fusion.

1/3 and 1/2 of their original numbers, respectively. Table 2 presents the number of load, store, cross-lane, and in-lane operations before arithmetic calculations for various vectorization methods across several kernels, demonstrating that Jigsaw outperforms the Multiple Loads and Multiple Permutations methods.

4 Evaluation

In this section, we evaluated the performance of *Jigsaw* scheme on both Intel and AMD architectures for varied classic stencils, which are widely used in applications.

4.1 Setup

Machines. We conducted experiments on two different hardware machines to evaluate Jigsaw and obtain results. The first machine consisted of two Intel Xeon Gold 6230R processors with 2.10 GHz clock speed, totaling 52 physical cores across two sockets. Each core contains a 32KB private L1 cache, a 1MB private L2 cache, and a unified 35.75MB L3 cache. Another platform is a Microsoft Azure high-performance node comprising a 2.45 GHz AMD EPYC 7V13 processor. It has 24 physical cores, each with a 768KB private L1 cache, a 12MB private L2 cache, and a unified 96MB L3 cache. Both platforms feature the AVX2 SIMD instruction set.

Kernels. Our experiments employed two distinct types of stencils, star and box, across different dimensions (1D, 2D, 3D) to ensure diversity, with specific parameters detailed in Table 3. Among them, Heat-1D, Heat-2D, and Heat-3D are the most commonly used basic kernels in stencil optimization research [8, 24, 51], which are 1D3P, 2D5P, and 3D7P star stencils, respectively. We fine-tuned the size and blocking of each stencil kernel based on relevant work to guarantee peak performance across all methods. These parameters are utilized in experiments involving parallel computation in conjunction with tiling techniques, as § 4.4 and 4.5. In contrast, § 4.2 and 4.3 delve deeper into the impact of varying problem sizes and iteration steps on performance. Consequently, a more diverse set of parameters is employed, as detailed in the respective subsections.

Table 3. Configuration for Stencil Benchmarks

Kernel	Points	Problem Size	Blocking Size
Heat-1D	3	10240000 × 10000	2000 × 1000
Star-1D5P	5	10240000 × 10000	2000 × 500
Star-1D7P	7	10240000 × 10000	2000 × 300
Heat-2D	5	10000 × 10000 × 10000	200 × 200 × 50
Star-2D9P	9	10000 × 10000 × 10000	200 × 200 × 25
Box-2D9P	9	10000 × 10000 × 10000	200 × 200 × 50
Heat-3D	7	256 × 256 × 256 × 1000	20 × 20 × 10
Box-3D27P	27	256 × 256 × 256 × 1000	20 × 20 × 10

Benchmarks. Experiments present a comprehensive analysis of Jigsaw scheme against classical vectorization algorithms (Auto Vectorization [52], Data Reorganization [61]), highly optimized DSLs (SDSL [24], Pluto [6]) and state-of-the-art optimization work (Folding [37], Tessellation [60]). Throughout all benchmarks, the OpenMP scheme was inherently supported for parallelization. Additionally, we used the GCC compiler version 11.2.1 and 9.4.0 on Intel and AMD platforms, respectively, with the "-O3 -mavx2" optimization flags enabled.

Matrices. Most work on stencil [10, 11, 39, 62–64] evaluate performance using GStencils/s, denoting the number of stencil points updated per second, as defined in Equation (3)

$$\text{GStencil/s} = \frac{T \times \prod_{i=1}^{n} N_i}{t \times 10^9} \tag{3}$$

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.

4.2 Ablation Study

In this subsection, we investigate how Jigsaw benefits from different optimizations. Figure 7 illustrates the performance improvements afforded by each optimization on AMD and Intel machines, taking the typical kernel Box-2D9P as an example. Moreover, Figure 8 illustrates the changes in data movement and calculation between vector registers before and after the application of SDF, also taking the Box-2D9P kernel as an example.

As depicted in Figure 7, the overall performance exhibits an upward trend with the increase in problem size and time iterations. Furthermore, as the input size and time iterations grow, the contribution of different optimization method tends to stabilize across both machines. Under various problem sizes and with larger time iterations (≥ 20), each optimization method demonstrates a considerable contribution.

Compared to the direct implementation using only the Tessellating Tiling [61] algorithm, the introduction of the LBV optimization resulted in performance improvements of 44.24% and 43.03% on AMD and Intel platforms, respectively. Subsequently, we introduced the SDF to eliminate vector-dimension conflicts in the 2D stencil. This method yielded a significant performance leap on AMD, achieving an improvement of 47.51%, and also provided a performance gain of 16.64% on Intel. This indicates that the method effectively reduced redundant data movements and minimized pipeline stalls. Following this, we introduced ITM to eliminate DAC in the time dimension and enhance computational density, which brought performance gains of 9.86% and 8.33% on

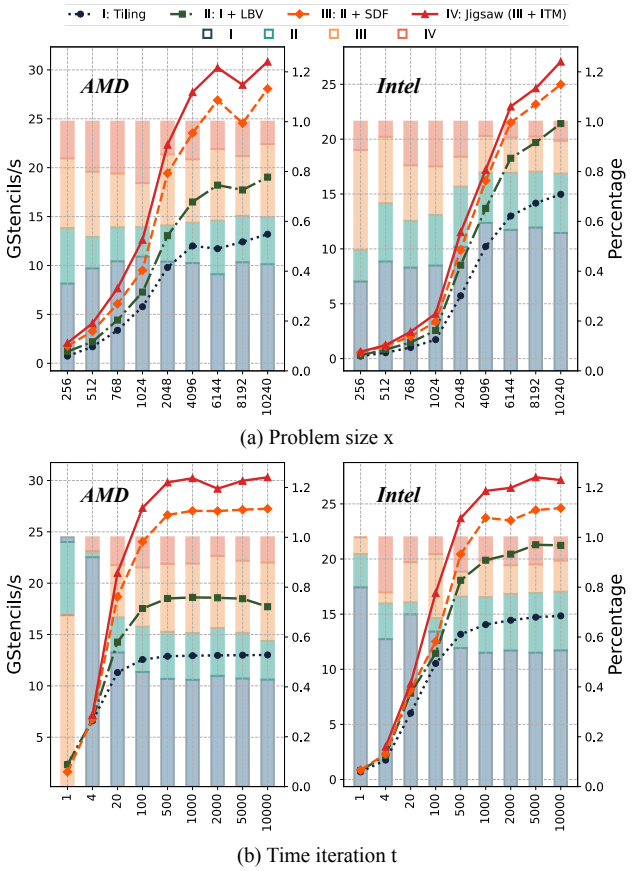

Figure 7. Performance Breakdown of Jigsaw. Figures (a) and (b) respectively depict the performance variation with problem size for a fixed number of time iterations and the performance variation with the number of time iterations for a fixed problem size. The line and bar respectively represent the absolute performance of different optimizations (left column) and their contribution to the Jigsaw method (right column).

AMD and Intel, respectively. At this stage, all optimizations within Jigsaw were fully implemented.

Figure 8 presents the statistical analysis of hotspot events in vectorization before and after the application of SDF optimization, measured by VTune[43]. As illustrated in the figure, the number of shuffles required during the vectorized computation process is significantly reduced following the implementation of SDF optimization, with a concomitant decrease in computation. This improvement is attributed to SDF's ability to eliminate a substantial amount of redundant vector-dimension conflicts, thereby optimizing the computational workflow and significantly reducing the proportion of data movement between vector registers in the computation process. Specifically, SDF reduces shuffle and computation time by 61.58% and 20.75%, respectively.

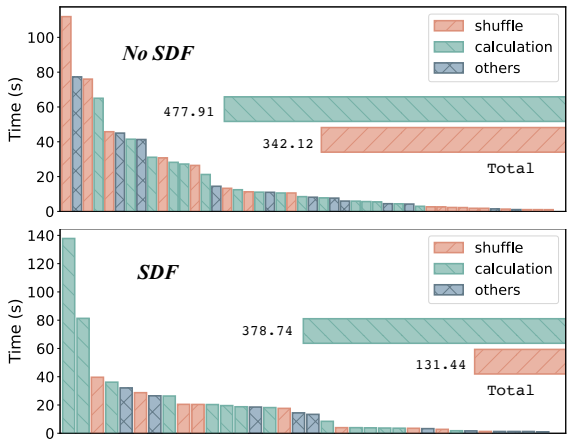

Figure 8. The impact of SDF on data movement and computation between registers. Vertical bars represent the ordered list of hotspot events with execution times exceeding 1s, while horizontal bars indicate the total time spent on shuffling and calculation throughout the entire vectorized execution process.

4.3 Sequential Block-free Results

In this subsection, we conducted sequential experiments without a tiling scheme to investigate the absolute performance of a single process across various sizes, in comparison with two classical vectorization methods. We selected four representative kernels that perform vectorized computations on a single thread, ranging from L1 cache to main memory. Figure 9 shows the comparison of our methods with others on AMD and Intel machines, respectively. The Auto Reorganization and Data Reorganization curves represent the *Multiple Loads* employed by the compiler and the *Multiple Permutations* technique, respectively. Jigsaw denotes the implementation that reduces DAC in the spatial dimension only (i.e., LBV+SDF), while T-Jigsaw represents the full-dimensional optimization incorporating ITM. This notation is consistently used in the subsequent § 4.4 and 4.5.

Star Stencil. The sequential performance results of Heat-1D and Heat-2D are presented in subfigures (a) and (b). As shown in Figure 9, our T-Jigsaw method outperforms other methods significantly on both machines, while Jigsaw also exhibits noticeable advantages in most cases. However, for tremendous problem size, the performance of Jigsaw and other methods tends to convergence due to the increasing cost of data transfer, which becomes the critical bottleneck of computation. For instance, on the AMD machine, as the problem size increases from the L1 cache to the memory hierarchy, all methods exhibit a similar trend of a stair-like decreasing curve, caused by memory bandwidth limitation.

Box Stencil. The other two subfigures illustrate the 2D9P and 3D27P Stencil. The Auto Vectorization and Data Reorganization methods introduce vector-dimension conflicts

Figure 9. Absolute sequential performance comparison in single-thread tiling-free on AMD and Intel machine.

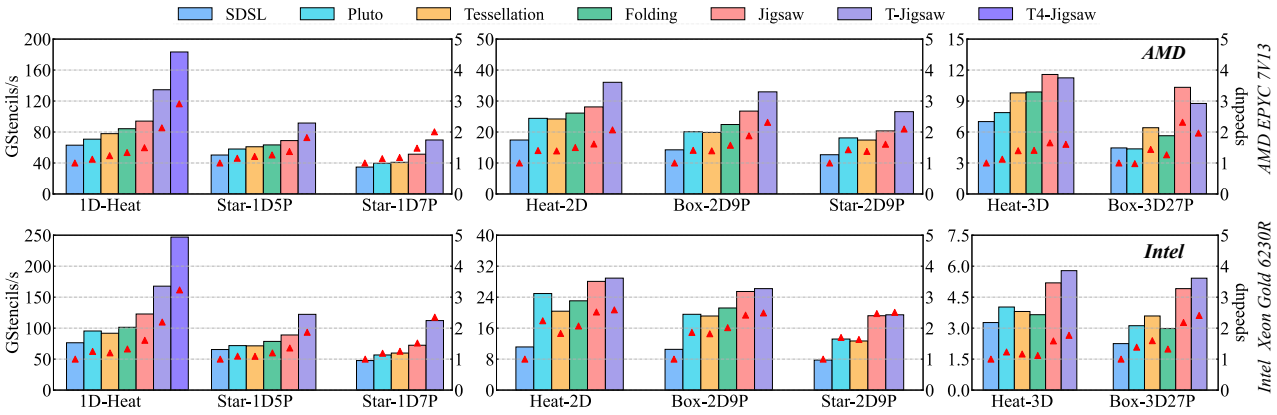

Figure 10. Performance and speedup comparison with cache-blocking on multicore AMD and Intel machine. The left column indicates absolute performance and the right shows the relative speedup ratio. The acceleration value for each method in each group is calculated relative to the lowest-performing method in that kernel, which is SDSL in this experiment.

on all spatial dimensions. Jigsaw's SDF method effectively addresses this issue and achieves significant visible performance improvements. However, the performance of T-Jigsaw method is not as good as Jigsaw's in the 3D box case, as shown in subfigure (d). It's due to ITM introducing too many data dependencies in 3D, which leads to an excess of register loading instructions that are no longer able to reduce the data transfer volume between cache and vector registers.

4.4 Parallel Cache-Blocking Results

In this subsection, we showcase the experimental integration of Jigsaw method with cache-tiling and parallelization schemes, validating the superiority of our approach. Specifically, we combined the Jigsaw vectorization with Tessellating Tiling [61] and compared it against SDSL [24], Pluto [8], Tessellation [60] and Folding (spatial) methods [37]. Figures 10 show the comparison with the baseline on two machines.

Taking all stencils with AVX2 instructions into account, our T-Jigsaw method exhibited remarkably improved performance compared to all reference work, achieved the speedup by an average of 2.148x on AMD and 2.466x on Intel, demonstrating significant benefits for the large-scale problem. From the figures, we can visually observe that the performance of stencil in all methods is related to the dimensionality (1D, 2D, 3D), shape (star, box), and order (1, 2, 3). With the increase in dimension, performance drastically declines due to the exponential growth of dependent grid points. In contrast, while shape and order also affect performance, they do not impose as severe a burden as dimensionality.

Compared with star stencil, our method exhibited greater advantages for box stencil (from 1.94x to 2.32x on AMD), benefiting from our SDF method, which can greatly eliminate the data preparation instructions and corresponding time required for multi-dimensional computation.

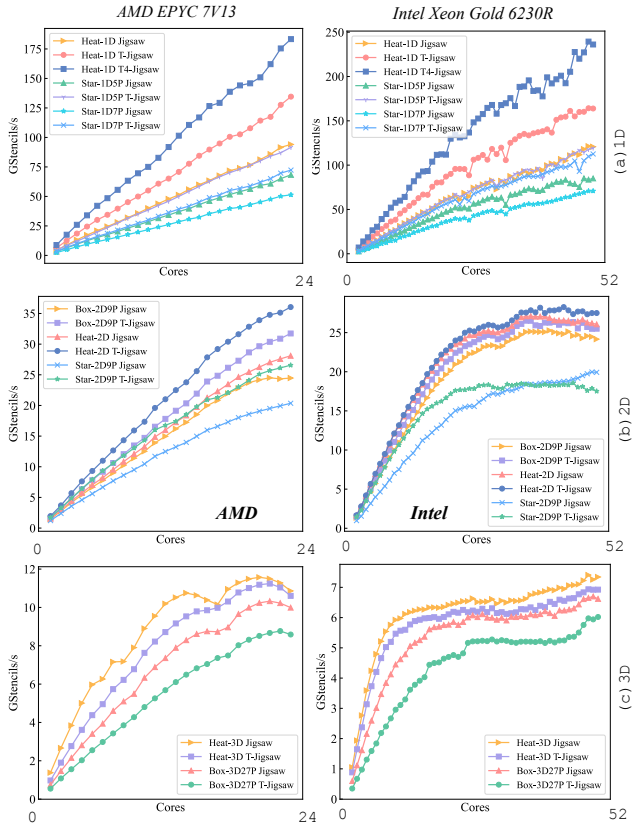

Figure 11. Scalability for stencils of various orders with different dimensions on multicore AMD and Intel machine.

Additionally, in the figure 10, T-4 Jigsaw represents the optimization achieved through a 4-step time fusion using ITM, corresponding to Figure 6. It can be observed that the introduction of multi-step fusion results in significant performance improvements (3.07x on average) in the Heat-1D stencil. However, in higher-dimensional stencils, multi-step fusion may introduce greater complexity, such as additional vector-dimension conflicts and register spills, which may not always contribute to performance enhancement. Therefore, we have applied this technique exclusively to the most effective 1D stencil.

4.5 Multi-Cores Scalability

We also evaluated the scalability of our Jigsaw and T-Jigsaw on two machines, conducting experiments ranging from one to all available cores of the processor. The experimental results were illustrated in Figure 11.

On the AMD platform, our approach exhibits consistent and excellent scalability, with almost linear scaling achieved for all kernels in 1D and 2D cases, and significant performance improvements obtained through the ITM strategy. However, the inherent complexity of multi-dimensional stencil computations causes a slight decline in scalability in the 3D case. Additionally, T-Jigsaw no longer retains the performance advantage over Jigsaw, primarily attributable to the heightened requirement for additional load operations during the computation of individual vectors. This degradation becomes more conspicuous when considering the scenario of a 3D box as opposed to a 3D star stencil.

On the Intel platform, we observed stable scalability growth in 1D, but different performance curves in the other two cases. To mitigate the impact of NUMA effects [34] - where processor attempts to access remote memory inevitably induce performance fluctuations due to mounting memory access latency - during scalability testing, for the growing cores, we alternately distribute them between NUMA1 and NUMA2. We observe that the scalability does not approximate linearity like the 1D case in higher dimensions, this discrepancy can be attributed to the inherent complexity of multi-dimensional stencil computations. Owing to its heightened data dependencies and poorer data locality, multi-dimensional stencil underutilized the cache, while also being more susceptible to bandwidth constraints. Furthermore, with the escalating number of cores, the inter-core communication emerges as a performance-limiting factor.

Moreover, the impact of stencil shape and order on performance is more vividly illustrated in Figure 11. Figure 11 (a) significantly demonstrates the impact of the order on performance, while Figures (b) and (c) show the effect of shape.

4.6 Discussion

In this subsection, we discuss the potential of the Jigsaw method for performance optimization across various instruction sets.

Given that all current AVX vector registers are physically composed of lanes [25], minimizing cross-lane communication is crucial for optimizing data transfer between registers. The LBV method effectively reduces this overhead by leveraging underlying architectural designs, yielding significant optimization benefits for AVX, AVX2, and AVX512. Regarding the newly developing instruction set, AVX10, which is poised to be a superset of the current AVX instruction sets and compatible with all existing instructions, we anticipate that LBV will also provide corresponding optimization benefits. Moreover, the SDF and ITM methods are inherently independent of vector register architecture, serving as universal optimization strategies. Therefore, we consider Jigsaw to be a general optimization design based on AVX features, with the potential to enhance performance across various instruction sets. However, only empirical experimentation will provide a definitive answer.

5 Related work

Optimization research on stencil computation has been extensively studied. The solutions can be broadly categorized

into three directions: improving computational performance, enhancing data reuse, and boosting data locality.

The compiler community has been engaged in researching general-purpose vectorization techniques [3, 22, 30, 31, 35, 42, 48]. Previous work has proposed solutions to the issue of DAC [17, 36, 57, 59]. DLT [23] is a landmark approach that addresses this problem by using a dimension-lifting transpose. However, its separation of spatial data into vl independent stencils makes it unfriendly to enhance data reuse. Folding [37] is one of the state-of-the-art approaches that addresses the overhead of data reorganization during vectorization, but it is limited to rank-1 matrices with parameter proportionality. Temporal vectorization [59] can perform vector calculations in the iteration space through vector register groups points with different time coordinates and is well applicable to the Gauss-Seidel stencils.

The pursuit of enhancing data reuse through the exploitation of stencil computation characteristics has garnered widespread attention. *Semi-Stencil* [14] optimize the iteration computation sequence by incorporating a gather pattern that alters the computational domain. Stock proposed a framework that enhances data locality and reduces register pressure by exploiting the associativity and commutativity [49]. Rawat introduced *LARS* [45] strategy, which flexibly schedules register usage. Detiz proposed a compiler optimization formula called Array Subexpression Elimination (ASE) to cope with common subexpressions that span cross kernel loop boundaries [15]. Similarly, Basu utilized the highly symmetric coefficients in stencil kernels to achieve data reuse via partial sums [7]. Yount presented the *YASK* [58] framework to vectorize points across the entire data space and generate high-performance code for 3D high-order stencils. Zhao [66–68] employed a greedy strategy to enhance instruction-level parallelism by exploiting block-level reuse. Moreover, Ahmad have recently proposed an FFT-based stencil computation method, offering a novel approach to enhancing the arithmetic intensity of stencil calculations [1, 2].

Tiling is a powerful technique to enhance data locality and facilitate cache reuse [26, 33, 53, 54]. In contrast to fine-grained parallelism in registers enabled by vectorization, titling produces a better coarse-grained parallelism at the cache level between tiles. Representative tiling techniques include Hyper-rectangle Tiling [16, 40, 44, 46], Time Skewing [28, 47, 55], Diamond Tiling [6, 8], Cache oblivious Tiling [18, 50, 51], Split Tiling [24], Hybrid Tiling [19] and Tessellating Tiling [61]. These tiling methods are mostly compiler techniques, and Wonnacott and Strout presented a comparison of the scalability of many existing tiling schemes [56]. Several automatic tuning frameworks [12, 21, 29, 65] have been proposed to accelerate stencil computations using tiling techniques. This paper integrated our vectorization method with tessellating tiling to attain optimal performance while simplifying implementation.

6 Conclusion

This paper proposes Jigsaw, a conflict-free stencil vectorization method to reduce DAC across all dimensions through tessellating swizzled registers with the finest-grained lanes. It comprises Lane-based Butterfly Vectorization, SVD-based Dimension Flattening and Iteration-based Temporal Merging, adeptly addressing DAC across spatial and temporal dimensions. Experimental results on different machines demonstrate that Jigsaw outperforms state-of-the-arts with promising speedup.

7 Acknowledgment

The authors sincerely thank the anonymous shepherd and reviewers for their valuable comments and allocated time towards improving this work. The work of Liang Yuan was supported in part by the National Natural Science Foundation of China under Grant No. 62372432, 62072431 and 62032023.

References

[1] Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Yimin Zhu. 2021. Fast Stencil Computations using Fast Fourier Transforms. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 8–21. https://doi.org/10.1145/3409964.3461803

[2] Zafar Ahmad, Mohammad Mahdi Javanmard, Gregory Croisdale, Aaron Gregory, Pramod Ganapathi, Louis-Noël Pouchet, and Rezaul Chowdhury. 2022. FOURST: A code generator for FFT-based fast stencil computations. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 99–108. https://doi.org/10.1109/ISPASS55109.2022.00010

[3] Randy Allen and Ken Kennedy. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.* 9, 4 (oct 1987), 491–542. https://doi.org/10.1145/29873.29875

[4] 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. https://doi.org/10.1145/1562764.1562783

[5] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[6] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.107

[7] Protonu Basu, Mary Hall, Samuel Williams, Brian Van Straalen, Leonid Oliker, and Phillip Colella. 2015. Compiler-Directed Transformation for Higher-Order Stencils. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 313–323. https://doi.org/10.1109/IPDPS.2015.103

[8] 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. https://doi.org/10.1145/1375581.1375595

[9] Diego Caballero, Sara Royuela, Roger Ferrer, Alejandro Duran, and Xavier Martorell. 2015. Optimizing Overlapped Memory Accesses in User-directed Vectorization. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 393–404. https://doi.org/10.1145/2751205.2751224

[10] 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. https://doi.org/10.1145/3295500.3356162

[11] 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. https://doi.org/10.1145/3627535.3638476

[12] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. 676–687. https://doi.org/10.1109/IPDPS.2011.70

[13] Standard Performance Evaluation Corporation. [n. d.]. SPEC2000. Accessed: 2024-03-26.

[14] Raúl de la Cruz and Mauricio Araya-Polo. 2014. Algorithm 942: Semi-Stencil. *ACM Trans. Math. Softw.* 40, 3, Article 23 (apr 2014), 39 pages. https://doi.org/10.1145/2591006

[15] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th International Conference on Supercomputing* (Sorrento, Italy) *(ICS '01)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/377792.377807

[16] Chris Ding and Yun He. 2001. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, Colorado) *(SC '01)*. Association for Computing Machinery, New York, NY, USA, 50. https://doi.org/10.1145/582034.582084

[17] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 82–93. https://doi.org/10.1145/996841.996853

[18] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing* (Cambridge, Massachusetts) *(ICS '05)*. Association for Computing Machinery, New York, NY, USA, 361–366. https://doi.org/10.1145/1088149.1088197

[19] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2018. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 66–75. https://doi.org/10.1145/2544137.2544160

[20] Intel Intrinsics Guide. 2015. URL: https://software. intel. com/sites/landingpage. *IntrinsicsGuide (18.05. 2018)* (2015).

[21] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. 2015. MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/2751205.2751223

[22] Mark Hampton and Krste Asanovic. 2008. Compiling for vector-thread architectures. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. Association for Computing Machinery, New York, NY, USA, 205–215. https://doi.org/10.1145/1356058.1356085

[23] 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: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*. Springer, 225–245. https://doi.org/10.1007/978-3-642-19861-8_13

[24] 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. https://doi.org/10.1145/2464996.2467268

[25] Intel. 2022. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) - Permuting Data Within and Between AVX Registers. (2022). https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-permuting-data-within-and-between-avx-registers-technology-guide

[26] F. Irigoin and R. Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 319–329. https://doi.org/10.1145/73560.73588

[27] 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. https://doi.org/10.1109/SC41404.2022.00035

[28] Guohua Jin, John Mellor-Crummey, and Robert Fowler. 2001. Increasing temporal locality with skewing and recursive blocking. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, Colorado) *(SC '01)*. Association for Computing Machinery, New York, NY, USA, 43. https://doi.org/10.1145/582034.582077

[29] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470421

[30] Ken Kennedy and John R. Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[31] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. 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, 127–138. https://doi.org/10.1145/2491956.2462187

[32] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 235–244. https://doi.org/10.1145/1250734.1250761

[33] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/106972.106981

[34] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue* 11, 7 (jul 2013), 40–51. https://doi.org/10.1145/2508834.2513149

[35] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[36] S. Larsen, E. Witchel, and S. Amarasinghe. 2002. Increasing and detecting memory address congruence. In *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. 18–29. https://doi.org/10.1109/PACT.2002.1105970

[37] 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. https://doi.org/10.1145/3458817.3476154

[38] Saeed Maleki, Yaoqing Gao, Maria J. Garzar´n, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 372–382. https://doi.org/10.1109/PACT.2011.68

[39] 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* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 199–211. https://doi.org/10.1145/3368826.3377904

[40] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC.2010.2

[41] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/1133981.1133997

[42] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/1454115.1454119

[43] Intel VTune Profiler. 2024. Intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler. Accessed: 2024-08-01.

[44] F. Rastello and T. Dauxois. 2002. Efficient tiling for an ODE discrete integration program: redundant tasks instead of trapezoidal shaped-tiles. In *Proceedings 16th International Parallel and Distributed Processing Symposium*. 8 pp–. https://doi.org/10.1109/IPDPS.2002.1016667

[45] 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. https://doi.org/10.1109/SC.2018.00049

[46] 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. https://doi.org/10.1109/SC.2000.10015

[47] Yonghong Song and Zhiyuan Li. 1999. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/301618.301668

[48] Narasimhan Sreraman and Ramaswamy Govindarajan. 2000. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming* 28 (2000), 363–400.

[49] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2594291.2594342

[50] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2010. Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. Association for Computing Machinery, New York, NY, USA, 49–59. https://doi.org/10.1145/1810085.1810096

[51] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1989493.1989508

[52] Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su. 2002. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal* 6, 1 (2002).

[53] Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449

[54] M. Wolfe. 1989. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing* (Reno, Nevada, USA) *(Supercomputing '89)*. Association for Computing Machinery, New York, NY, USA, 655–664. https://doi.org/10.1145/76263.76337

[55] David Wonnacott. 2002. Achieving scalable locality with time skewing. *International Journal of Parallel Programming* 30 (2002), 181–221.

[56] David G Wonnacott and Michelle Mills Strout. 2013. On the scalability of loop tiling techniques. *IMPACT 2013* 3 (2013).

[57] P. Wu, A.E. Eichenberger, and A. Wang. 2005. Efficient SIMD code generation for runtime alignment and length conversion. In *International Symposium on Code Generation and Optimization*. 153–164. https://doi.org/10.1109/CGO.2005.18

[58] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK—Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. 30–39. https://doi.org/10.1109/WOLFHPC.2016.08

[59] Liang Yuan, Hang Cao, Yunquan Zhang, Kun Li, Pengqi Lu, and Yue Yue. 2021. Temporal vectorization for stencils. 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 82, 13 pages. https:

//doi.org/10.1145/3458817.3476149

[60] 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. https://doi.org/10.1145/3337821.3337835

[61] 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. https://doi.org/10.1145/3126908.3126920

[62] 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 ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 167–179. https://doi.org/10.1145/3577193.3593705

[63] 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 ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 251–263. https://doi.org/10.1145/3577193.3593716

[64] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 53, 17 pages. https://doi.org/10.1109/SC41406.2024.00059

[65] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 155–164. https://doi.org/10.1145/2259016.2259037

[66] 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. https://doi.org/10.1145/3295500.3356210

[67] 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. https://doi.org/10.1145/3437801.3441598

[68] 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. https://doi.org/10.1109/P3HPC.2018.00009

[69] Hao Zhou and Jingling Xue. 2016. A Compiler Approach for Exploiting Partial SIMD Parallelism. *ACM Trans. Archit. Code Optim.* 13, 1, Article 11 (mar 2016), 26 pages. https://doi.org/10.1145/2886101