LoRAStencil: Low-Rank Adaptation of Stencil Computation on Tensor Cores

Yiwei Zhang*†¶#, Kun Li†‖#, Liang Yuan‡‖, Jiawen Cheng§, Yunquan Zhang‡, Ting Cao†, Mao Yang†

*Institute of Computing Technology, Chinese Academy of Sciences
and University of Chinese Academy of Sciences, Beijing, China
†Microsoft Research, Beijing, China
‡Chinee Academy of Sciences, Beijing, China
§Tsinghua University, Beijing, China

***Abstract*—Stencil computations play a pivotal role in numerous scientific and industrial applications, yet their efficient execution on specialized hardware accelerators like Tensor Core Units (TCUs) remains a challenge. Despite previous attempts to address this issue, performance bottlenecks persist, particularly in memory access redundancy. This paper introduces LoRAStencil[1], a novel stencil computing system designed to mitigate memory access redundancies on TCUs through low-rank adaptation. We first identify a nuanced form of this redundancy, dimension residue, specific to TCUs. Then LoRAStencil leverages orchestrated mathematical transformations to decompose stencil weight matrices into smaller rank-1 matrices, facilitating efficient data gathering along residual dimensions. It comprises three key components: memory-efficient Residual Dimension Gathering to facilitate more data reuse, compute-saving Pyramidal Matrix Adaptation to exploit the inherent low-rank characteristics, and performance-boosting Butterfly Vector Swapping to circumvent all data shuffles. Comprehensive evaluations demonstrate that LoRAStencil address dimension residues effectively, which outperforms state-of-the-arts with up to a 2.16x speedup, offering promising advancements for efficient tensorized stencil computation on TCUs by Low-Rank Adaptation.**

***Index Terms*—Stencil Computation, High Performance Computing, Matrix Multiplication, Tensor Cores, GPU**

I. Introduction

Deep learning represents a crucial stride towards achieving human-level intelligence, with matrix multiplication (MM) operations playing a central and accelerated role through specialized units in current and emerging processors. On NVIDIA GPUs, these units, known as Tensor Core Units (TCUs), significantly enhance the execution of various deep learning models [1].

In contrast to the relatively standardized MM operations in deep learning models, computational patterns within High-Performance Computing (HPC) exhibit greater complexity and diversity. The Berkeley view abstracts frequently-used computational patterns as 13 different dwarfs in HPC, each with numerous branches catering to various scientific and industrial applications [2, 3].

¶Work done during an internship at Microsoft Research (in 2022.10), with Project Lead (kunli@microsoft.com).
#Both authors contributed equally.
‖Corresponding author.
[1]LoRAStencil is available at https://github.com/HPHEX/LoRAStencil.

Stencil computation, acknowledged as one of the 13 HPC dwarfs, represents a performance-limiting algorithm addressing scientific and industrial problems such as fluid dynamics [4, 5], weather forecasting [6, 7], earth modeling [8] and wave equation [9, 10]. Characterized by numerous kernels, stencil computation iteratively updates each point within a d-dimensional spatial grid along the temporal dimension by the weighted sum of itself and neighboring points [11, 12].

For an extended period, research in HPC and deep learning has increasingly diverged, each focusing on specific application domains. Much of the prior work on stencil computations has been centered on CPU or GPU CUDA cores, thus failing to leverage the wealth of algorithms and newly-released hardware like TCUs in deep learning. One pioneering effort is ConvStencil [13], which represents the first attempt to bridge the gap between stencil computations in HPC and convolution operations in deep learning. Through a well-structured data layout known as stencil2row, ConvStencil transforms stencil computation into MM, thereby leading the way in unlocking the power of TCUs on GPU architectures.

In MM-based computing systems like ConvStencil, a significant yet hard-to-see performance bottleneck emerges. Architecturally, the smallest computing unit (fragment) on TCU hardware is designed for two-dimensional MM operations. During memory access, data are loaded into the TCU fragment in 2D tiles. However, during computation, the inner product operation in MM can only gather data along a single dimension (either columns or rows) on each matrix. When the dimensionality of the stencil kernel exceeds one, the number of dimensions involved in each computation is fewer than that for memory access. This discrepancy results in *dimension residue* specific to the TCU architecture, where redundant loads and computations along the residual dimension contribute to a low compute-to-memory ratio.

Dimension residue is exemplified in the design of ConvStencil, where the stencil2row transformation is employed to prepare the data layout for subsequent MM operations. However, this transformation introduces two additional matrices with a significant overlap of repeating elements. As a consequence, ConvStencil incurs a notable increase in memory access and storage overhead, exacerbating the memory-bound nature inherent in stencil computations.

In this paper, we propose **Lo**w-**R**ank **A**daptation Stencil (LoRAStencil), a stencil computing system designed to reduce memory access redundancies across both dimensions on TCUs through a series of mathematically-equivalent low-rank matrix adaptation.

The design of LoRAStencil is grounded on two important observations: 1) A rank-1 matrix can be expressed as the outer product of a column vector $\boldsymbol{u}$ and a row vector $\boldsymbol{v}$; 2) The matrix constituted by stencil weights in fact reside on a low intrinsic rank-r, allowing it to be decomposed into r rank-1 matrices.

Guided by these observations, the key idea behind LoRAStencil is *Low-Rank Adaptation*. Leveraging the rank-1 weight matrices characterized by the lowest rank, LoRAStencil initiates weight-data gathering along a single dimension through MM. Considering that a rank-1 weight matrix can be decomposed into the outer product of two vectors, it is theoretically feasible to extend data gathering to residual dimensions via mathematical transformations. For a stencil with an arbitrary rank weight matrix, the original weight matrix can be decomposed into the sum of several low-rank rank-1 weight matrices. Consequently, accumulating the results involving multiple rank-1 weight matrices facilitates the computation of the entire stencil.

LoRAStencil incorporates three key techniques: memory-efficient Residual Dimension Gathering, compute-saving Pyramidal Matrix Adaptation, and performance-boosting Butterfly Vector Swapping.

Residual Dimension Gathering is centered around Matrix Chain Multiplication on the TCU fragment. Rank-1 matrices, which can be expressed as the outer product of two vectors, enable continuous weight-data gathering along the residual dimensions. This computational approach enhances memory access efficiency by facilitating more effective data reuse on the TCU, thereby reducing both the volume and latency of data loading and alleviating the memory-bound nature inherent in stencil computations.

Pyramidal Matrix Adaptation extends the algorithm from rank-1 matrices to matrices of any rank. It effectively transforms the original weight matrix in stencil computation into multiple overlapped rank-1 weight matrices. Additionally, by leveraging the symmetry often found in stencil computations, it efficiently exploits the inherent low-rank characteristics of the weight matrix, thereby reducing the number of generated rank-1 weight matrices and minimizing redundant computations on the TCU.

Butterfly Vector Swapping operates at the vector level, employing a butterfly-like weight swapping algorithm to meticulously organize and control computations on the TCU fragment. Essentially, it optimizes the computational process of the algorithm from a mathematical perspective, circumventing the need for a plethora of time-consuming data shuffles within the TCU fragment. This optimization streamlines the implementation of Matrix Chain Multiplication algorithms, effectively minimizing idle bubbles in computations.

In comparison to several state-of-the-art approaches on GPU (cuDNN [14, 15], Brick [16–18], DRStencil [19]) and TCU (AMOS [20], TCStencil [21] and ConvStencil [13]), experimental results validate the effectiveness of LoRAStencil.

Our contributions are outlined as follows:

- We introduce LoRAStencil, an innovative stencil computing system designed to reduce redundancies in memory access across both dimensions on Tensor Core Units through a series of mathematically-equivalent low-rank matrix adaptation.
- LoRAStencil consists of 3 key techniques: 1) memory-efficient Residual Dimension Gathering on rank-1 matrix transformation to address dimension residues; 2) compute-saving Pyramidal Matrix Adaptation from rank-1 matrices to complete stencil computation; 3) performance-boosting Butterfly Vector Swapping at the vector level for an efficient mapping from algorithm to hardware.
- We implement these techniques and generalize them on various kernels. Experimental results demonstrate the effectiveness of LoRAStencil, surpassing various state-of-the-arts and achieving up to 2.16x speedup.

II. Background and Motivation

Algorithm 1 Box-2D9P stencil (1 time step)

```
Input: mesh $A$, weight $c_{11} \sim c_{33}$
Output: mesh $B$
1: for point[i][j] in $A$ do
2:     $B[i][j] = c_{11} \times A[i-1][j-1] + c_{12} \times A[i-1][j] + c_{13} \times A[i-1][j+1] + c_{21} \times A[i][j-1] + c_{22} \times A[i][j] + c_{23} \times A[i][j+1] + c_{31} \times A[i+1][j-1] + c_{32} \times A[i+1][j] + c_{33} \times A[i+1][j+1]$
3: end for
```

Stencil computations perform iterative updates on multi-dimensional inputs following a predefined pattern, with the set of points involved determined by the *radius*, also referred to *order*. The shape of the predefined pattern is primarily classified into two types: *star* and *box*. A star stencil computes the weighted sum of a central point and its neighbors that are displaced in only a single dimension. In contrast, a box stencil involves calculating the weighted sum within a square (or cube) centered on the central point. Each point in stencil computation is associated with specific weights. Algorithm 1 illustrates the stencil computation of Box-2D9P (radius: 1, shape: box, dimensions: 2, points: 9), which updates in a single temporal iteration.

A. Tensor Cores

Tensor Core Units on NVIDIA GPUs represent a specialized hardware component, meticulously engineered to expedite matrix multiplication and accumulation (MMA), as delineated in Equation 1. These units demonstrate a performance significantly superior to that of traditional CUDA cores when executing MMA operations.

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

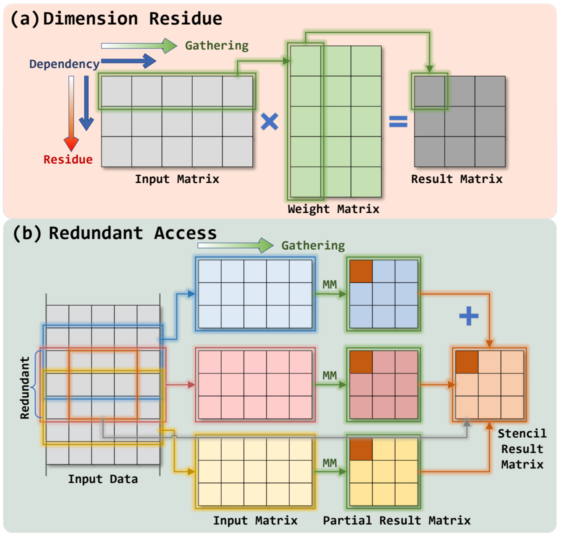


Fig. 1: (a) Dimension Residue Problem in stencil computations on TCU. (b) Redundant memory access in the residual dimension.

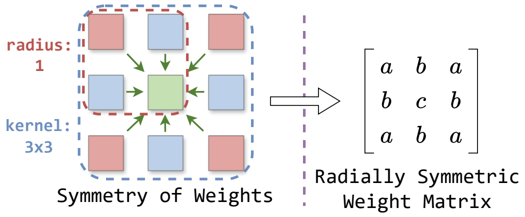
TCUs can be programmed at the warp level through the CUDA Warp Matrix Multiply and Accumulate (WMMA) API for conducting matrix operations. Before being processed by TCUs, data must be loaded into the fragment - a specialized data structure. The dimensions of fragments adhere to strict specifications, with current support in FP64 precision accommodating only $8\times8\times4$ MMA operations (referring to $m = 8$, $n = 8$ and $k = 4$ in Equation (1)).

B. Dimension Residue

Dimension Residue is a critical bottleneck in tensorized stencil computation, stemming from the discrepancy between MM and stencil dependency. We illustrate this issue with a straightforward example.

Figure 1(a) illustrates the misalignment between 2D stencil dependency dimensions and MM computational dimensions. In stencil computations, dependencies exist across both dimensions. However, the atomic operation in MM is the inner product of vectors, which only gather weight-data pairs along a single dimension. As depicted in Figure 1(a), the horizontal dimension represents the collected dimension, while the vertical dimension is the residual dimension (where dependencies are not collected).

Figure 1(b) demonstrates the redundant access caused by dimension residue. In tensorized stencil computation for Box-2D9P, a 3×3 square (orange block) is to be computed, which necessitates reliance on a 5×5 square. It can be observed that the input is read three times to form three 3×5 matrices. Each matrix undergoes MM to collect horizontal dependencies, and then partial result matrices are accumulated to collect vertical dependencies, yielding the stencil computation result. During this process, the data within red input matrix is also read by blue and yellow input matrices, resulting in significant redundant access.



$$\begin{bmatrix} a & b & a \\ b & c & b \\ a & b & a \end{bmatrix}$$

Fig. 2: Symmetry of Weights and Radially Symmetric Matrix. The same color indicates that the corresponding points have identical weights.

C. Symmetry in Stencil

In industrial and scientific computing, stencil computations with constant weights and regular grid points are widely employed [11, 22–24]. In such stencils, the weights frequently exhibit symmetric properties [12, 25–30], meaning that neighboring points with the same Euclidean distance of their corresponding dependence share identical weight. This characteristic is particularly evident in the solutions of practical problems such as heat conduction [11, 31, 32] and wave propagation equations [8, 33], where the weights inherently possess such symmetric traits.

Taking the Box-2D9P stencil as an example, Figure 2 illustrates the symmetric properties of these weights, where the same color indicates that neighboring points share identical weights. We refer to the matrix composed of such stencil weights as a radially symmetric matrix. Few studies have also explored the symmetry in stencil computations [12, 26, 34], however, they have not delved into the rank-related properties of the weight matrices. We observe that, due to the row and column symmetry inherent in radially symmetric matrices, these matrices exhibit low-rank characteristics. Specifically, for a kernel with a radius of h, and a weight matrix W of size $(2h + 1) \times (2h + 1)$, it holds that $\text{rank}(W) \leq h + 1$. We can exploit this low-rank property to convert stencil into efficient matrix computations on TCUs.

D. Mathematical Derivation

In this subsection, we will mathematically introduce the computational process of LoRAStencil using inductive reasoning. Through mathematical derivation, we provide a novel Matrix Chain Multiplication approach to tensorized stencil computation. We elucidate our derivation by transitioning from the specific case ($\text{rank} = 1$) to the general case ($\text{rank} > 1$). The explanation of symbols used in our analysis is listed in Table I.

From Algorithm 1, we can observe that stencil computation is essentially the accumulation of the element-wise product of two matrices, which is the Frobenius inner product fundamentally. For instance, for a stencil with a kernel size of n^2, where $n = 2h+1, h \in \mathbb{N}^+$, to compute the result at point $(h+1, h+1)$, the computation can be derived from Equation 2,

$$\langle C, X\rangle_{\text{F}} = \sum_{i=1}^{2h+1}\sum_{j=1}^{2h+1} c_{i,j}x_{i,j} \tag{2}$$

where C and X respectively denote the weight matrix and input matrix.

1) $\mathbf{rank = 1}$. When the weight matrix C is a rank-1 matrix, C contains only one linearly independent column vector, denoted as $\boldsymbol{u}$. Consequently, C can be represented through the basis vector $\boldsymbol{u}$ and another vector $\boldsymbol{v}$, which implies

$$C = \boldsymbol{u} \otimes \boldsymbol{v}^{\mathbf{T}} \tag{3}$$

Consequently, substituting Equation (3) into (2), we can transform the original stencil computation, which is in the form of the Frobenius inner product, into a vector-matrix-vector multiplication, like Equation (4):

$$\langle C, X \rangle_{\mathrm{F}} = \sum_{i=1}^{n} \sum_{j=1}^{n} u_i x_{i,j} v_j = \boldsymbol{u}^{\mathbf{T}} X \boldsymbol{v} \tag{4}$$

thus, we have obtained a new stencil computation method for updating a single point at once.

TCUs offers us an opportunity to efficiently compute stencil for multiple elements simultaneously through tensorization. Assuming the data size for single tensorized stencil computational update is $m \times m$, the size of input matrix X then is $(m + 2h) \times (m + 2h)$ to satisfy dependencies. The weight matrix U has dimensions $m \times (m+2h)$, and the weight matrix V is $(m + 2h) \times m$. The matrices U and V are respectively derived from expansion of vectors $\boldsymbol{u}^{\mathbf{T}}$ and $\boldsymbol{v}$. The elements in U and V are as follows:

$$u_{i,j} = \begin{cases} u_{j-i+1}, & i \leq j \leq i + 2h \\ 0, & others \end{cases} \tag{5}$$

$$v_{i,j} = \begin{cases} v_{i-j+1}, & j \leq i \leq j + 2h \\ 0, & others \end{cases} \tag{6}$$

Thus, the vector-matrix-vector product of p-th row of U ($\boldsymbol{u_p}$), matrix X,and q-th column of V ($\boldsymbol{v_q}$) is the stencil result at the point $(p+h, q+h)$. Therefore, for a stencil with a weight matrix of C, its tensor computation can be transformed into the **Matrix Chain Multiplication (MCM)** of U, X, and V, specifically:

$$\langle C, X \rangle_{\mathrm{FT}} = UXV \tag{7}$$

where $\langle C, X \rangle_{\mathrm{FT}}$ represents the tensorized stencil computation of input X with weight C.

TABLE I: Notation

Notation	Definition
W	Original weight matrix
C	Original/Decomposed rank-1 weight matrix
$\boldsymbol{u}/\boldsymbol{v}$	Rank decomposition vector of matrix C
U/V	Weight matrix of vector $\boldsymbol{u}/\boldsymbol{v}$ expansion
X	Input data matrix
h	Radius of the stencil kernel
n	Edge length of the stencil kernel
m	Edge length of the data for once tensor calculation
r	Rank of original weight matrix W
$\langle C, X \rangle_{\mathrm{F}}$	Scalar stencil computation of input X with weight C
$\langle W, X \rangle_{\mathrm{FT}}$	Tensor stencil computation of input X with weight W

2) $\mathbf{rank > 1}$. In case where the rank of the weight matrix is not equal to 1, let us denote the weight matrix as W with $\mathrm{rank}(W) = r$, where $r \in [1, n]$. Based on the SVD of matrices, W can be decomposed into r matrices each with a $\mathrm{rank} = 1$. This implies:

$$W = \sum_{k=1}^{r} C_k = \sum_{k=1}^{r} \boldsymbol{u_k} \otimes \boldsymbol{v_k}^{\mathbf{T}} \tag{8}$$

Then, we transform the stencil computation of the weight matrix W into a sum of stencil computations of rank-1 weight matrices C. By combining this with Equation (7), we can further transform it to the sum of a series of MCM.

$$\langle W, X \rangle_{\mathrm{FT}} = \sum_{k=1}^{r} \langle C_k, X \rangle_{\mathrm{FT}} = \sum_{k=1}^{r} U_k X V_k \tag{9}$$

Hence, the tensorized stencil with a rank-r weight matrix W can be transformed into the sum of r MCM of U, X, and V. This transforms the stencil computation pattern from the Frobenius inner product to the matrix computation on TCUs.

III. LoRAStencil

In this section, we will provide a detailed introduction to LoRAStencil, incorporating examples. This includes three key techniques: Residual Dimension Gathering, Pyramidal Matrix Adaptation, and Butterfly Vector Swapping.

A. Overview

We begin by providing an overview of the LoRAStencil for general stencils, as illustrated in Figure 3. Similar to the mathematical derivation subsection (II-D), we first introduce the handling of the LoRAStencil for the specific case $\mathrm{rank} = 1$. Subsequently, we extend this discussion to encompass the general case where $\mathrm{rank} > 1$.

$\mathbf{rank = 1}$. Residual Dimension Gathering (**RDG**, Subsection III-B) transforms rank-1 stencil computations into MCM on the TCU, thereby addressing the redundant memory accesses caused by Dimension Residue. Initially, the weight matrix C is rank-decomposed into the outer product of vectors $\{\boldsymbol{u}, \boldsymbol{v}\}$, these vectors are then mapped onto fragments to construct the corresponding weight matrices $\{U, V\}$. Subsequently, by performing RDG on the weight matrices $\{U, V\}$ and the input matrix X, the final stencil result is obtained.

$\mathbf{rank > 1}$. Pyramidal Matrix Adaptation (**PMA**, Subsection III-C) is the key technique that extends LoRAStencil to general stencils, leveraging low-rank properties of weight matrix to reduce redundant computations. This process splits W into the sum of rank-1 weight matrices C of decreasing sizes. Each C_i is then subjected to aforementioned rank-1 process to obtain their respective partial result matrices P_i. By summing up all P_i, the final stencil result can be obtained.

Butterfly Vector Swapping (**BVS**, Subsection III-D) is the core step for achieving efficient MCM on the TCU. By applying mathematical transformations to the mapping functions, BVS overcomes the challenge posed by TCU hardware's natural inefficiency in handling MCM, eliminating redundant data movement among threads.

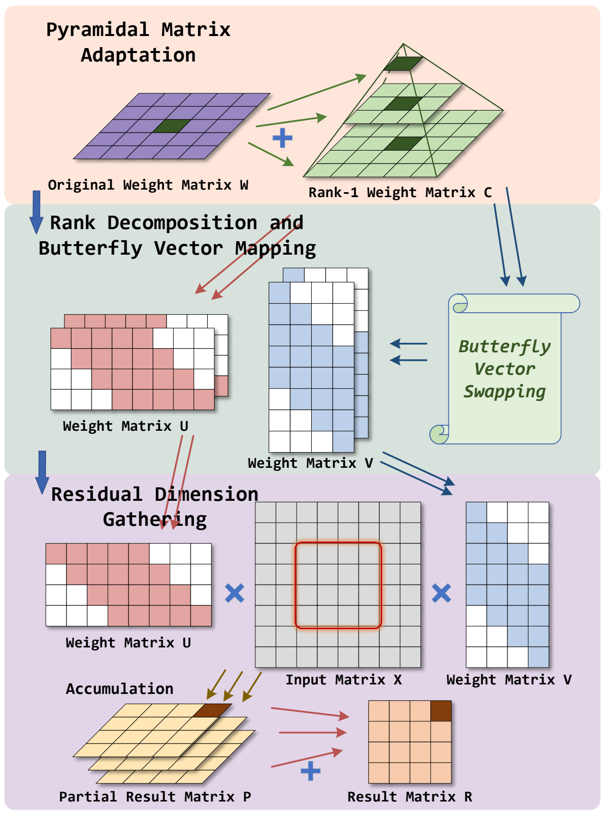


Fig. 3: Overview of the LoRAStencil Computational Process for General Stencils. LoRAStencil comprises three core components: Residual Dimension Gathering, Pyramidal Matrix Adaptation, and Butterfly Vector Swapping.

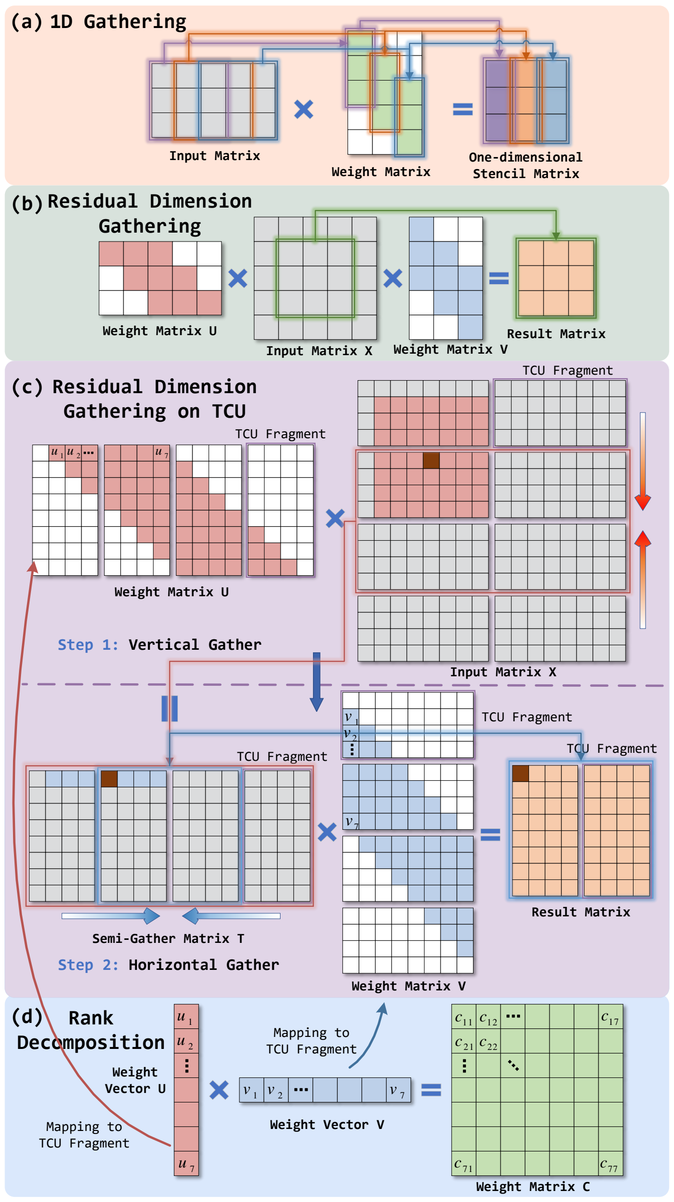


Fig. 4: Residual Dimension Gathering and Rank Decomposition of LoRAStencil.

B. *Residual Dimension Gathering*

In this subsection, we introduce a new stencil computation method, *Residual Dimension Gathering*, aimed at maximizing the reduction of redundant memory accesses across all dimensions.

The development of RDG is predicated upon two key observations: 1) Through the strategic arrangement of the weight matrix, it is feasible to gather single-dimensional dependencies for multiple adjacent points without redundant memory access. As illustrated in Figure 4(a), the MM of the input matrix with weight matrix (where green blocks represent weights and white represent zero) yields horizontal dependencies collection for three adjacent data sets (represented by purple, orange and blue, respectively), with horizontal access redundancy-free during this computation. 2) When the weight matrix is of rank-1, its dependencies can be transformed into the product of a column vector and a row vector, as discussed in Section II-D. This implies that vertical dependencies can be collected first, followed by the gathering of horizontal dependencies.

Based on above observations, we extend the single-dimensional non-redundant gathering to all dimensions, resulting in RDG. RDG enables redundancy-free gathering across all dimensions for tensorized stencil computation, including residual dimension, through MCM. As illustrated in Figure 4(b), 2D stencil computation for the central segment of matrix X is accomplished by aggregating dependencies in both orientations through the MCM of weight matrix U, input matrix X, and weight matrix V (where white blocks represent zero).

RDG Process. Figure 2(c) demonstrates the specific computation process of RDG on TCU using a 7×7 stencil kernel as an example, which encompasses two steps: vertical gather and horizontal gather.

In Step 1, the weight matrix U is multiplied by the input matrix X, yielding the semi-gather matrix T that aggregates vertical dependencies. Each element within matrix T is the cumulative sum of weights u associated with its vertically adjacent elements. We will introduce the input matrix and the weight matrix respectively.

The weight matrix U, a 8×16 matrix (including 4 fragments), is employed for the vertical gathering of X. In matrix U, the red and white blocks denote weight and zero elements, respectively. U originates from the expansion and mapping of the weight vector $\boldsymbol{u}$ across the 8×4 fragment. The vector $\boldsymbol{u}$ serves as the basis column of the weight matrix C, obtained through the rank decomposition of the rank-1 matrix C, as illustrated in Figure 4(d). Every row in U contains n non-zero elements, equivalent to the kernel's edge length, with the layout of each row being a one-position rightward shift from the preceding row. The general construction method for the weight matrix has been elucidated in Subsection II-D. Here, we provide the mapping function for a 7×7 stencil kernel:

$$u_{i,j} = \begin{cases} u_{j-i}, & i+1 \le j \le i+7 \\ 0, & others \end{cases} \tag{10}$$

where $i \in [1, 8], j \in [1, 16]$. Fragments in matrix U that align in vertical sequence with the horizontal sequence of matrix X are subject to MMA. The outcomes are subsequently accumulated at the corresponding positions in a semi-gather matrix T. This process requires a total of 8 MMA operations.

The input matrix X (depicted in gray) is a 16×16 square matrix load from shared memory without redundancy, utilized to compute stencil results of the central 8×8 square. Each element of X resides within a 4×8 fragment, as determined by the right-multiplication with TCU. The dark brown block within matrix X denotes the first point in the result matrix, while red blocks indicate dependencies of the dark brown.

In Step 2 of the horizontal gather, dependencies across all dimensions are collected to yield the final result matrix. This process is similar to Step 1 and simply involves MM of the semi-gather matrix T with the weight matrix V to complete the MCM of U, X, and V. Within T, it is observable that the vertical dependencies in X are compressed into a single blue row, leaving only horizontal dependencies to be addressed. In matrix V, the blue blocks denote the expansion of the weight vector $\boldsymbol{v}$, while the white blocks, as in U, represent zero elements. Since V is required to gather horizontal dependencies, the weight vector $\boldsymbol{v}$ is arranged vertically within the fragment, within its layout as specified in Equation (11):

$$v_{i,j} = \begin{cases} v_{i-j}, & j+1 \le i \le j+7 \\ 0, & others \end{cases} \tag{11}$$

where $i \in [1, 16], j \in [1, 8]$. In Step 2, 4 MMA operations are conducted, totaling 12 MMA operations required in RDG.

In RDG, MCM is employed to reduce redundant access across all dimensions, thereby enhancing data reuse. This method fully capitalizes the characteristics of MM and tensor data structure, facilitating a seamless integration of stencil computation with TCU hardware. Furthermore, RDG does not require data layout transformation, thus circumventing the necessity to construct redundant matrices in shared memory, a step that is indispensable in ConvStencil. This reduction in memory demand enables an increased number of threads to function concurrently on streaming multiprocessors (SMs), thereby improving hardware occupancy and parallelism.

Redundancy Reduction Analysis. For the RDG approach, when performing tensorized stencil computations with a kernel radius of h, ideally, the size of the matrix updated at once is $2h \times 2h$, striking a balance between optimal data reuse and register occupancy. However, due to the limitation of TCU fragment size 4×8 in FP64, there will be at least $\lceil 2h/8 \rceil$ fragments horizontally and $\lceil 2h/4 \rceil$ fragments vertically. Thus, the number of grid points updated at once is $32\lceil h/2 \rceil \lceil h/4 \rceil$. For the $a \times b$ input, the computation must be executed $\frac{ab}{32\lceil h/2 \rceil \lceil h/4 \rceil}$ times, each load $2\lceil 2h/8 \rceil \times 2\lceil 2h/4 \rceil$ fragments to satisfy the dependency. Consequently, the requisite number of fragments to be loaded from shared memory can be calculated by Equation (12),

$$\text{RDG} = \frac{ab}{8} \tag{12}$$

While in ConvStencil, to compute $8 \times (2h+2)$ consecutive elements, it requires loading $2 \times \lceil (2h+1)^2/4 \rceil$ fragments from shared memory [13]. The total number is calculated by (13),

$$\text{ConvStencil} = 2 \times \lceil \frac{(2h+1)^2}{4} \rceil \times \lceil \frac{a}{16(h+1)} \rceil b \tag{13}$$

Given the substantial dimensions of a and b, the ratio of memory access volume between ConvStencil and RDG is articulated as Equation (14).

$$\frac{\text{ConvStencil}}{\text{RDG}} = \frac{1}{h+1} \lceil \frac{(2h+1)^2}{4} \rceil \tag{14}$$

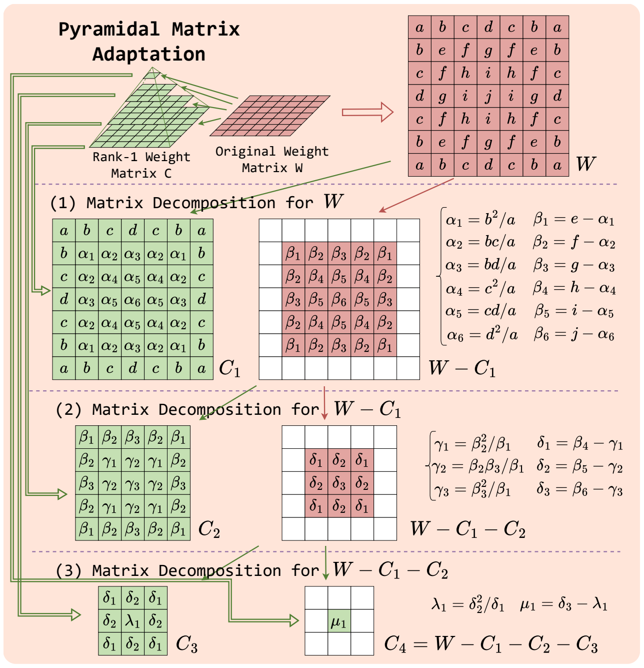
In the case of Box-2D49P stencil (where $h = 3$), the memory access volume of ConvStencil is 3.25x that of RDG. This indicates that RDG eliminates 69.23% of redundant memory accesses in ConvStencil. When $h = 4$, the redundancy of ConvStencil escalates to 4.2x that of RDG, with the latter method eliminating 76.19% of the redundant accesses.

C. Pyramidal Matrix Adaptation

After introducing the RDG computation for the rank-1 weight matrix stencils, the subsequent challenge addressed is the extension of the aforementioned method to the general weight matrix W. In Subsection II-D, we have mathematically derived that: for any weight matrix W with $\text{rank}(W) = r$, the stencil result is equivalent to the sum of stencil performed individually by rank-1 weight matrix C_i, where $W = \sum_{i=1}^{r} C_i$, as delineated in Equations (9).

Moreover, we observe that stencil weights commonly exhibit radial symmetry in scientific and industrial computations, as discussed in II-C. The low-rank properties of stencil weights offer us an opportunity for redundant computation reduction. Leveraging this characteristic, we propose the *Pyramidal Matrix Adaptation* method for efficiently computing general stencils on TCUs.

In PMA, we leverage the radial symmetry of the stencil weights to decompose the original W into the sum of pyramid-shaped rank-1 weight matrices C. As described in II-C, for a kernel with radius h and side length $n = 2h+1$, the rank of its



Pyramidal Matrix Adaptation
Rank-1 Weight Matrix C
Original Weight Matrix W
(1) Matrix Decomposition for W
$\alpha_1 = b^2/a$ $\beta_1 = e - \alpha_1$
$\alpha_2 = bc/a$ $\beta_2 = f - \alpha_2$
$\alpha_3 = bd/a$ $\beta_3 = g - \alpha_3$
$\alpha_4 = c^2/a$ $\beta_4 = h - \alpha_4$
$\alpha_5 = cd/a$ $\beta_5 = i - \alpha_5$
$\alpha_6 = d^2/a$ $\beta_6 = j - \alpha_6$
C_1
$W - C_1$
(2) Matrix Decomposition for $W - C_1$
$\gamma_1 = \beta_2^2/\beta_1$ $\delta_1 = \beta_4 - \gamma_1$
$\gamma_2 = \beta_2\beta_3/\beta_1$ $\delta_2 = \beta_5 - \gamma_2$
$\gamma_3 = \beta_3^2/\beta_1$ $\delta_3 = \beta_6 - \gamma_3$
C_2
$W - C_1 - C_2$
(3) Matrix Decomposition for $W - C_1 - C_2$
$\lambda_1 = \delta_2^2/\delta_1$ $\mu_1 = \delta_3 - \lambda_1$
C_3
$C_4 = W - C_1 - C_2 - C_3$

Fig. 5: Pyramidal Matrix Adaptation of LoRAStencil. The red matrices represent the original and radially symmetric matrices, while the green matrices denote the rank-1 matrices.

weight matrix is at most $h+1$. Here, we assume $\mathrm{rank}(W) = h+1$. By employing matrix elementary transformation, we can decompose W into $h+1$ rank-1 matrices C.

Figure 5 illustrates the specific process of PMA using the Box-2D49P stencil as an example, which shows how to recursively decompose a radially symmetric matrix W into a sum of rank-1 matrices. In Figure 5, the red matrices represent the original matrix W and the decomposed radially symmetric matrices, while the green matrices represent the rank-1 matrices C obtained from the matrix decomposition.

Firstly, for the original matrix W, we can construct a rank-1 matrix C_1, where the i-th row of C_1 is $w_{i,1}/w_{1,1} \times w_{1,:}$. Thus, C_1 is the outer product of vector $\boldsymbol{u}$ and vector $\boldsymbol{v}$, where

$$\boldsymbol{u} = \frac{1}{w_{1,1}} \cdot [w_{1,1}, w_{2,1}, ..., w_{7,1}]^{\mathrm{T}}, \boldsymbol{v} = w_{1,:}$$

Since W is a radially symmetric matrix, it exhibits both row and column symmetry. Therefore, $\boldsymbol{v}$ is a symmetric vector, i.e., $w_{1,1} = w_{1,7}, w_{1,2} = w_{1,6}, ...$; similarly, $\boldsymbol{u}$ is also a symmetric vector. Consequently, the rank-1 matrix C_1 obtained from the outer product of $\boldsymbol{u}$ and $\boldsymbol{v}$ is also a radially symmetric matrix. Additionally, C_1 share the same first and last row with W, and the first and last element of the middle rows of C_1 are identical to those in W. Consequently, the matrix $W - C_1$ yields a 5×5 radially symmetric matrix, as illustrated in Figure 5(1).

Subsequently, applying the aforementioned matrix decomposition to the $W - C_1$ yields a 5×5 rank-1 matrix C_2 and a 3×3 radially symmetric matrix $W - C_1 - C_2$. By recursively executing this process, we can ultimately decompose W into the sum of rank-1 matrices C_1, C_2, C_3 and C_4, where C_4 is a 1×1 scalar matrix. Each C can be factorized as the outer product of vectors $\{\boldsymbol{u}, \boldsymbol{v}\}$.

Finally, by applying RDG to each C_i individually with the input matrix, and subsequently accumulating the partial result matrix, we can obtain the final result for the stencil with arbitrary weights. It is noteworthy that the weight matrix C_4 comprises solely one point, signifying that the update point is dependent only on itself, without reliance on its surrounding neighbors. Thus, no additional MM is required to obtain the corresponding partial result matrix.

We can generalize the aforementioned process to symmetric stencils with arbitrary rank. In general, for a radially symmetric matrix W with $\mathrm{rank}(W) = h+1, h \in \mathbb{N}^+$, we can use the PMA method to decompose it into $h+1$ rank-1 matrices C with decreasing dimensions, as illustrated in Equation (15).

$$\begin{aligned} W_{(2h+1)\times(2h+1)} &= (C_1)_{(2h+1)\times(2h+1)} + \cdots + (C_{2h+1})_{1\times1} \\ &= \sum_{i=1}^{h+1} (C_i)_{(2h+3-2i)\times(2h+3-2i)} \end{aligned} \tag{15}$$

Consequently, by employing PMA, we have extended RDG to the computation of general kernel stencils. Since the input matrix utilized for each RDG in PMA remains constant, fragments of input data are amenable to reuse, ensuring that the PMA process does not introduce redundant memory access.

Quantitative Performance Analysis. In the preceding section, we demonstrated the advantage of LoRAStencil in terms of memory access. Next, we quantitatively analyze the number of computational instructions (MMA) required.

As mentioned in Subsection III-B, when executing tensorized stencil computation with a kernel radius h, it requires $\frac{ab}{32\lceil h/2\rceil\lceil h/4\rceil}$ computations, with each involving h RDG operations. Each RDG involves the MCM of matrix U, X and V. $U \times V = T$ include $2\lceil h/2\rceil \times 2\lceil h/4\rceil$ MMA, while $T \times V$ involves $2\lceil h/2\rceil$ MMA. Therefore, the number of MMA operations required by LoRAStencil is given by Equation (16),

$$\mathrm{LoRAStencil} = 2h\lceil\frac{h}{2}\rceil(2\lceil\frac{h}{4}\rceil + 1) \times \frac{ab}{32\lceil h/2\rceil\lceil h/4\rceil} \tag{16}$$

While in ConvStencil, there is an absence of fragment reuse, thus the number of required MMA operations is equivalent to the count of data load instructions, as delineated in Equation (13) of Subsection III-B.

In Box-2D49P stencil ($h = 3$), the ratio of the number of MMA operations required by LoRAStencil to ConvStencil is $36/26$, approximately 1.38. Compared to the significantly reduced memory accesses, the increase in computational workload is small. The denser computation also allows the TCU computational power to be more fully exploited, which has not yet reached its bottleneck. Experimental results further demonstrate our superior performance.

D. Butterfly Vector Swapping

Through the PMA and RDG method, LoRAStencil can efficiently perform tensor computations for general stencil kernels. However, the core operation of RDG is MCM, which is

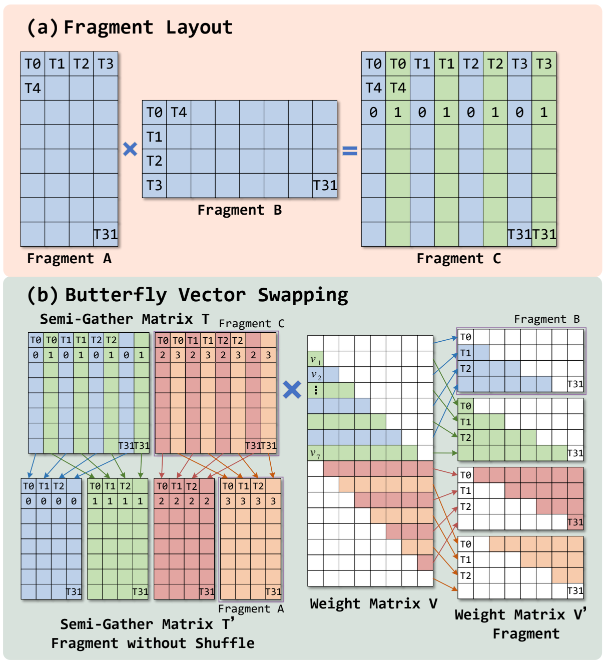


Fig. 6: Fragment Layout on TCU and Butterfly Vector Swapping of LoRAStencil. The same color indicates data residing in the same register across different threads.

not natively supported by WMMA interfaces. Worse still, due to its unique layout within fragments at FP64 precision, the accumulator matrix obtained after MMA operations cannot be directly used for subsequent MMA. Additional data movement is required to execute MCM, which impedes the computational process and becomes the critical bottleneck for performance. To achieve efficient MCM, we propose the *Butterfly Vector Swapping* method to address the issue through mathematical equivalent transformation.

In TCU MMA operations, the matrix size and data layout are strictly defined. For instance, in FP64 precision, the left-multiplied, right-multiplied and accumulator matrix are stored in fragments A, B and C with dimensions 8×4, 4×8 and 8×8, respectively. As WMMA operations are executed at the warp level, with each warp comprises 32 threads, each thread holds 1 element from fragment A and B. Similarly, each thread holds 2 elements from fragment C, stored in register 0 (R0) and register 1 (R1), for ease of reference. The layout of elements in the fragment and their corresponding thread registers is illustrated in Figure 6(a), where R0 and R1 are indicated by blue and green blocks, respectively. It can be observed that two consecutive elements ($\{(0,0),(0,1)\}$) in C are stored within the same thread (T0). To perform MM, we need to split fragment C into two fragments A. However, the direct mathematical partitioning of fragment C ($1\sim4$ columns and $5\sim8$ columns form two fragment A, respectively) introduces shuffle operations among threads. For instance, transferring the first green column from R1 of threads $\{0,4,..,28\}$ to R0 of threads $\{1,5,...,29\}$ (like the layout in fragment A) introduces inter-thread shuffling and impedes computation.

We address this issue by mathematically butterfly-swapping the rows of the right-multiplied matrix at the vector level. As shown in Figure 6(b), when performing horizontal gather ($T\times V$), the layout of T does not support MM, and directly splitting T, as discussed earlier, would introduce inter-thread shuffling. By observing the MM, we note that when the columns in T and the rows in V undergo the same order of vector swapping, the result remains unchanged, as shown in Equation (17),

$$\begin{aligned}\sum_{k=1}^{n}\boldsymbol{t_k}\boldsymbol{v_k}^{\mathbf{T}} &=[\cdots\boldsymbol{t_i}\cdots\boldsymbol{t_j}\cdots]\times[\cdots\boldsymbol{v_i}^{\mathbf{T}}\cdots\boldsymbol{v_j}^{\mathbf{T}}\cdots]^{\mathrm{T}}\\ &=[\cdots\boldsymbol{t_j}\cdots\boldsymbol{t_i}\cdots]\times[\cdots\boldsymbol{v_j}^{\mathbf{T}}\cdots\boldsymbol{v_i}^{\mathbf{T}}\cdots]^{\mathrm{T}}\end{aligned} \tag{17}$$

where

$$\boldsymbol{t_i}=[t_{1,i},t_{2,i},...,t_{n,i}]^{\mathrm{T}},\boldsymbol{v_i}=[v_{i,1},v_{i,2},...,v_{i,n}]^{\mathrm{T}}$$

respectively denote the column and row vector of T and V.

Thus, we can directly use elements located in the same register within T (same color) to construct fragment A, while performing the corresponding butterfly-row-vector swapping in weight matrix V. Specifically, since the columns $\{1,3,5,7\}$ of matrix T form the first four columns (fragment A) of T', it suffices to map the rows $\{1,3,5,7\}$ of matrix V to the first four rows (fragment B) of V' to ensure $TV=T'V'$. As illustrated in Figure 6(b), constructing the left-multiplied fragment T' is achievable without inter-thread shuffling, and when multiplied with the corresponding butterfly-row-swapping right-multiplied fragment V', it accomplishes the MCM in RDG.

This transformation only requires modifying the mapping function from the weight vector $\boldsymbol{v}$ to matrix V' (fragment B), without introducing additional physical data movement or computational operations, thereby circumventing data shuffling among threads. Following this optimization, we overcome the hardware design challenge with RDG, achieving efficient MCM operations on TCU.

IV. Optimization

A. Kernel Fusion

LoRAStencil can be applied to any stencil kernel. However, due to the limitations of fragment size, some small kernels struggle to effectively utilize TCU. Therefore, we temporally fuse some kernels to enhance computation within TCUs. For instance, in Figure 7, updating an 8×8 matrix (orange squares) requires loading eight 4×8 fragments. Box-2D9P only utilizes 10×10 green elements, wasting the outer three rows/columns of gray elements. By temporally 3x fusing it into Box-2D49P, we can reduce fragment storage waste by $96/156\approx61.54\%$, thereby improving the utilization of TCUs.

B. Asynchronous Data Copy

Before loading data from shared memory into the fragment, it necessitates the data copy from global memory to shared memory. Direct data copy introduces register occupancy, signifying that data must traverse through global memory, registers, and then shared memory. Fortunately, the Ampere architecture introduces asynchronous feature from

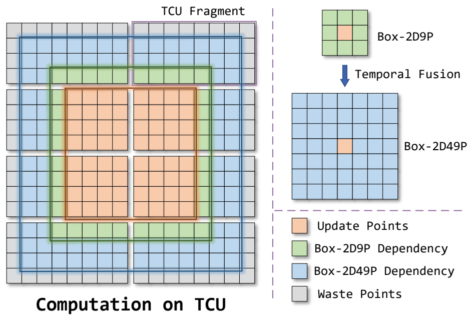


Fig. 7: Kernel fusion of LoRAStencil.

global memory to shared memory, which circumvents the overhead of transferring data through registers and is beneficial in saving the usage of intermediate registers [35]. Therefore, we efficiently accomplish data copy from global memory to shared memory by employing the `cp.async` instruction through PTX (Parallel Thread Execution) embedded assembly.

C. Generalization

After introducing 2D LoRAStencil, it can be straightforwardly generalized to other dimensions. For 1D stencil, as it only has dependency in a single dimension, one MM suffices to complete the gathering process. There is no redundancy in dimension orthogonal to the update direction, hence MCM is unnecessary.

LoRAStencil can be mathematically extended to support higher-dimensional tensor architectures with ease. For 3D stencil, since TCU fragments physically only support two dimensions and do not support 3D data storage and computation operations, we decompose the 3D stencil into multiple 2D planes and perform LoRAStencil computations on each plane. For planes with a single weight in 3D star-type stencils, we utilize CUDA cores for computation, providing more opportunities for parallel utilization of both computing units on the GPU.

Algorithm 2 The 3D Algorithm of LoRAStencil.

```
Input: $h$ : radius of the stencil kernel, $A_1, ..., A_{2h+1}$ : input data of plane
    $\{1, ..., 2h+1\}$, $W_1, ..., W_{2h+1}$ : weight matrix of plane $\{1, ..., 2h+1\}$
Output: $B$ : output matrix of 3D stencil
 1: function LoRAStencil3D
 2:     for $i \leftarrow 1$ to $2h+1$ do
 3:         if $W_i$ contain only one weight then
 4:             // CUDA Core for Point-Wise Computation
 5:             $B$ += CUDACoreCompute($A_i, W_i$)
 6:         else
 7:             // Tensor Core for 2D LoRAStencil
 8:             $B$ += TensorCoreCompute($A_i, W_i$)
 9:         end if
10:     end for
11: end function
```

Algorithm 2 illustrates the 3D stencil computation process of LoRAStencil. A 3D stencil kernel with a radius h can be viewed as the superposition of $2h+1$ 2D planes. When the i-th plane contains only a single weight, it indicates that there is no dependency collection from other neighboring points in that layer. Consequently, we can utilize CUDA Cores to perform direct point-wise multiplication and accumulation operations on the data and the corresponding weight. Conversely, when the i-th plane requires multiple dependency points, Tensor Cores are employed to execute matrix computations, specifically the 2D LoRAStencil calculations. Notably, for a regular 3D stencil, each plane represents a star-shaped or box-shaped 2D stencil. Therefore, when the plane contains more than one grid point, it includes at least five points (star-2D5P), necessitating the use of TCUs for 2D LoRAStencil. Finally, by aggregating the results of these $2h + 1$ planes, we obtain the 3D stencil result for the h-th layer.

V. Evaluation

A. Experimental Setup

Machine. Our experimental platform comprises an AMD EPYC 7V13 processor and an NVIDIA A100 Tensor Core GPU. The A100 GPU is configured with 80GB of HBM2e memory, featuring a 5120-bit memory interface width, and offers a memory bandwidth of 1935GB/s. The A100 GPU is equipped with 108 SMs, each containing 4 Tensor Cores. The TCUs deliver a peak FP64 performance of 19.5 TFLOPS.

State-of-the-arts. We compare LoRAStencil with various state-of-the-arts for a comprehensive analysis, including cuDNN [14, 15], AMOS [20], Brick [16–18], DRStencil [19], TCStencil [21] and ConvStencil [13].

cuDNN and AMOS represent advanced endeavors in the computation of convolutions. Given that convolutions and stencils share an intrinsic similarity in the computational patterns, a comparison with these benchmarks is undertaken.

Notably, the design of TCStencil is specifically tailored for computation on TCUs at FP16 precision. Given that the TCStencil algorithm design on specific fragments, and the size and shape of fragments at FP16 and FP64 differ on the TCU, it is not feasible to directly convert TCStencil to FP64 precision. For this scenario, we adopt the same analysis as in ConvStencil [13]. On the A100 TCU, FP16 computation speed is 16 times faster than FP64. And under the same memory bandwidth, FP16 read/write speed is 4 times faster than FP64. Hence, in the best-case scenario, the speed of TCStencil in FP64 would be a quarter of FP16. Therefore, in our evaluation, we divide the TCStencil speed by 4 for comparison.

ConvStencil performs 3x temporal fusion for small kernels, a technique equally employed in LoRAStencil. Therefore, LoRAStencil does not gain an unfair advantage in this respect. Thus, through performance comparison with ConvStencil, there is no need to further elucidate whether the performance edge of LoRAStencil is attributable to kernel fusion.

Benchmarks. We use various stencil kernels with different shapes as benchmarks. The specifics are detailed in Table II, including five star kernels (Heat-1D, 1D5P, Heat-2D, Star-2D13P, and Heat-3D) and three box kernels (Box-2D9P, Box-2D49P, and Box-3D27P), sourced from [13, 29].

Metrics. Most work on stencil evaluate performance using $\mathrm{GStencil/s}$ (Gigastencils per second, denoting the number of

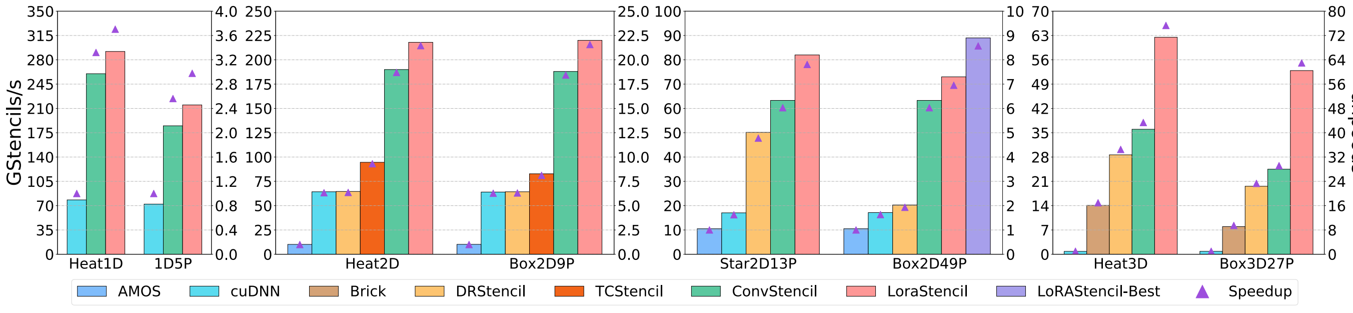


Fig. 8: Performance comparison of LoRAStencil with state-of-the-art approaches on GPU and TCU. LoRAStencil-Best represents the performance of LoRAStencil when the original weight matrix is a rank-1 matrix.

stencil points updated per second) as a metric. We also adopt this metric, as defined in Equation (18)

$$\text{GStencil/s} = \frac{T \times \prod_{i=1}^{n} N_i}{t \times 10^9} \tag{18}$$

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.

B. State-of-the-art Comparison

Figure 8 illustrates the performance comparison of LoRAStencil with all state-of-the-arts, where the left and right vertical axes represent absolute performance and relative speedup, respectively. The speedup value for each method is calculated relative to the lowest-performing method in that kernel. It is evident that LoRAStencil demonstrates significantly improved performance compared to all reference works.

It can be observed that all works optimized for stencil outperform cuDNN and AMOS. This is attributed to the lack of specialized optimizations for stencil within both cuDNN and AMOS frameworks. Additionally, cuDNN does not employ TCU for acceleration, while although AMOS utilizes TCU, it does not optimize the mapping from stencil to TCU, squandering a significant portion of computational power. Compared to cuDNN and AMOS, LoRAStencil achieves an average speedup of 20.11x and 14.45x, respectively.

In works tailored optimized for stencils, LoRAStencil also exhibits significant performance improvements. Compared to Brick, DRStencil, and TCStencil, LoRAStencil achieves an average performance acceleration of 5.54x, 2.82x, and 2.48x, respectively. Compared to the state-of-the-art ConvStencil, LoRAStencil improves the performance sustainably by 1.12x on minimum, 2.16x on maximum and 1.37x on average. Notably, in 3D, the performance improvement is particularly pronounced. This is attributed to the implementation of LoRAStencil, which is not constrained by kernel size, meaning that it maintains high utilization of TCU fragments even with small kernels, and can update multiple grid points at once. In contrast, ConvStencil suffers from poor fragment utilization with small kernels due to the specific matrix construction of the algorithm, and can only update fewer grid points at once calculation. Consequently, ConvStencil is compelled to undertake a 3x temporal fusion in 3D, which greatly increases the number of dependencies. This exacerbates memory-bound characteristics, issues such as register overflow and insufficient shared memory become more severe, while also introducing excessive computational overhead, severely limiting performance.

TABLE II: Configuration for Stencil Benchmarks

Kernel	Points	Problem Size	Blocking Size
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

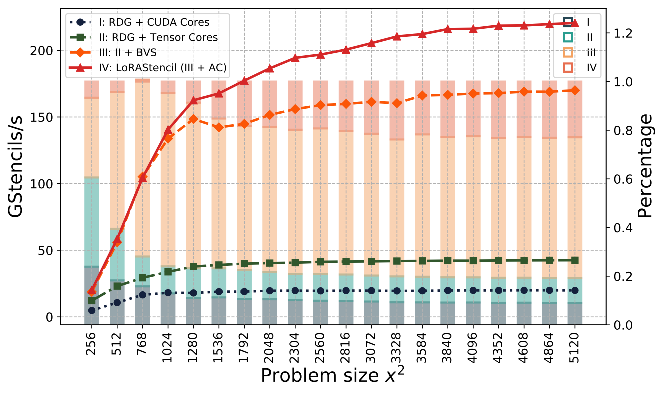


Fig. 9: Performance breakdown of LoRAStencil.

C. Performance Breakdown

In this subsection, we investigate how LoRAStencil benefits from different optimizations. Figure 9 illustrates the performance improvements afforded by each optimization method, taking the typical kernel Box-2D9P as an example.

As depicted in Figure 9, the contributions of different optimizations gradually stabilize with increasing input size. Compared to the direct use of RDG in CUDA Cores, the introduction of Tensor Cores leads to a performance enhance-

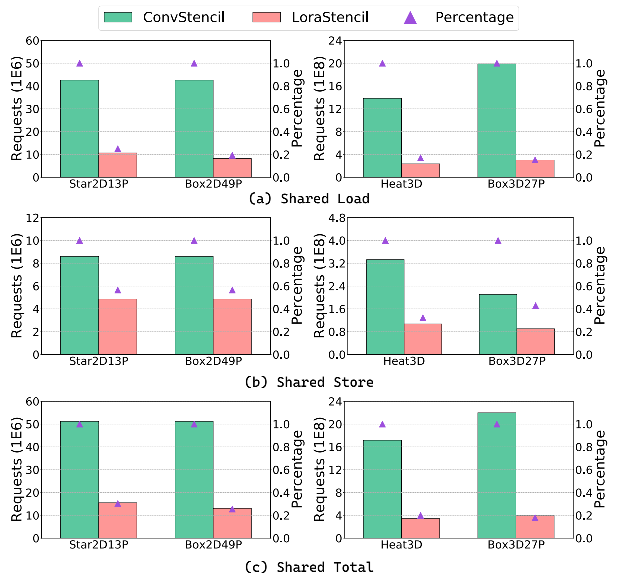


Fig. 10: Comparison of load, store, and total requests in shared memory between ConvStencil and LoRAStencil.

ment of 2.14x. This improvement is attributed to the powerful MM capabilities of TCU. Subsequently, we employed BVS to eliminate inter-thread shuffle operations introduced during MCM. It is evident that this strategy led to a significant performance leap, achieving a 4.00x speedup. This indicates that inter-thread shuffling is a critical bottleneck in previous implementation, and BVS effectively addresses this issue mathematically, reducing computational bubbles and facilitating efficient MCM. Finally, we introduced asynchronous copy (AC) operations, eliminating the reliance on intermediate registers, resulting in a performance gain of 29.7%. With this, we have completed all optimizations in LoRAStencil.

D. *Memory and Compute Comparison with ConvStencil*

In this subsection, we will conduct a more detailed comparison with ConvStencil in terms of memory and computation to illustrate the effectiveness and superiority of the approach. We exemplify our analysis by four representative kernels, Star2D13P, Box2D49P, Heat3D and Box3D27P, which across different dimensions. Figure 10 shows the actual shared memory loads, stores and total requests during computation by all warps, as measured by Nsight Compute [36].

LoRAStencil significantly reduces the number of shared memory load requests, with the average decreasing to 19.1%

TABLE III: Compute comparison to ConvStencil.

Kernel	Box-2D49P		Box-3D27P	
Metrics	CT[1]	AI[2]	CT	AI
ConvStencil	69.97%	3.59	36.88%	1.65
LoRAStencil	86.42%	7.41	49.31%	2.53

[1] CT denotes the Compute (SM) Throughput percentage
[2] AI denotes the Arithmetic Intensity (FLOP/byte).

of ConvStencil. This substantial reduction in redundant fragment loads can be attributed to two main factors. Firstly, the RDG approach diminishes the redundant loads in the residual dimension. Secondly, the computational pattern of LoRAStencil facilitates extensive reuse of registers in warp, further minimizing the necessity for repeated memory accesses.

Regarding shared stores, which refer to the instructions for copying data from global memory to shared memory, the number of store requests in LoRAStencil averages at 47.0% of that in ConvStencil. This efficiency is attributed to LoRAStencil's elimination of the need for additional data layout transformation, which ConvStencil requires for constructing two stencil2row matrices. These matrices occupy more shared memory, reducing the maximum number of threads that can work simultaneously and thus lowering the hardware occupancy. The total number of requests is reduced by 76.6% in LoRAStencil compared to ConvStencil.

Table III shows the comparison of the computational throughput (CT) and arithmetic intensity (AI). The throughput reports the achieved percentage of utilization with respect to the theoretical maximum, and AI denotes the amount of computation accomplished per data access. LoRAStencil exhibits an enhancement in both the utilization of TCU and AI, enabling closer to the peak performance capabilities of the hardware.

VI. Related Work

Optimization research on stencil computation has been extensively studied [2, 3]. Representative works can largely be categorized into two directions based on architecture.

On CPU, vectorization is a crucial pathway to enhancing computational performance [12, 22, 37, 38]. Data Layout Transformation (DLT) [24, 39], stands as a milestone, adeptly addressing the data alignment conflicts introduced during SIMD instruction computations. Utilizing stencil computing features to enhance data reuse has also been a subject of numerous studies [26, 40, 41]. Blocking, or tiling, is a powerful technique to enhance data locality and facilitate cache reuse [42–45]. Representative tiling techniques in stencil work include rectangle tiling [23, 46, 47], time skewing [48–50], diamond tiling [29, 51, 52], cache-oblivious tiling [11, 53], and tessellating tiling [54].

On GPU, stencil optimization has also been widely researched [55–58]. Tiling on GPUs includes spatial tiling [17, 59, 60] and temporal tiling [61–63]. Other stencil optimizations include prefetching [64], unrolling [65], and streaming [66]. Brick [16–18], by capitalizing on data reuse within fine-grained blocks of stencil computations, reduces prefetching and cache pressure, proposing a universal framework across CPU and GPU. DRStencil [19] accelerates low-order stencil computations through the fusion-partition optimization techniques and implements an effective code generation framework. These works are all aimed at optimization for CUDA cores. As tailored for TCU stencil optimization, to our knowledge, only TCStencil [21] and ConvStencil [13] exist. TCStencil pioneers in mapping stencil computation to hardware-accelerated matrix computation, but is limited to FP16 precision and is plagued by dimension residue. ConvStencil combines the similarities between convolution and stencil, forging a bridge between HPC and deep learning. However, it also fails to address dimension residue and incurs additional memory overhead.

VII. Conclusion

This paper proposes LoRAStencil, a stencil computing system designed to mitigate memory access redundancy on TCUs through low-rank adaptation. It comprises Residual Dimension Gathering, Pyramidal Matrix Adaptation and Butterfly Vector Swapping, adeptly addressing the issue of Dimension Residue. The experiment results show that LoRAStencil outperforms state-of-the-arts with promising performance speedup.

References

[1] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[3] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[4] H. Huynh, Z. J. Wang, and P. E. Vincent, "High-order methods for computational fluid dynamics: A brief review of compact differential formulations on unstructured grids," *Computers & fluids*, vol. 98, pp. 209–220, 2014.

[5] D. J. Lusher, S. P. Jammy, and N. D. Sandham, "Opensbli: Automated code-generation for heterogeneous computing architectures applied to compressible fluid dynamics on structured grids," *Computer Physics Communications*, vol. 267, p. 108063, 2021.

[6] Y. Ao, C. Yang, X. Wang, W. Xue, H. Fu, F. Liu, L. Gan, P. Xu, and W. Ma, "26 pflops stencil computations for atmospheric modeling on sunway taihulight," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 535–544.

[7] T. Ben-Nun, L. Groner, F. Deconinck, T. Wicky, E. Davis, J. Dahm, O. D. Elbert, R. George, J. McGibbon, L. Trümper *et al.*, "Productive performance engineering for weather and climate modeling with python," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–14.

[8] M. Jacquelin, M. Araya-Polo, and J. Meng, "Scalable distributed high-order stencil computations," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–13.

[9] K. Akbudak, H. Ltaief, V. Etienne, R. Abdelkhalak, T. Tonellot, and D. Keyes, "Asynchronous computations for solving the acoustic wave propagation equation," *The International Journal of High Performance Computing Applications*, vol. 34, no. 4, pp. 377–393, 2020.

[10] L. Qu, R. Abdelkhalak, H. Ltaief, I. Said, and D. Keyes, "Exploiting temporal data reuse and asynchrony in the reverse time migration," *The International Journal of High Performance Computing Applications*, vol. 37, no. 2, pp. 132–150, 2023.

[11] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, 2011, pp. 117–128.

[12] K. Li, L. Yuan, Y. Zhang, and Y. Yue, "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*, 2021, pp. 1–15.

[13] Y. Chen, K. Li, Y. Wang, D. Bai, L. Wang, L. Ma, L. Yuan, Y. Zhang, T. Cao, and M. Yang, "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*, 2024, pp. 333–347.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[15] Nvidia, "cudnn," https://developer.nvidia.com/cudnn, accessed: 2024-03-26.

[16] T. Zhao, S. Williams, M. Hall, and H. Johansen, "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)*. IEEE, 2018, pp. 59–70.

[17] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen, "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*, 2019, pp. 1–44.

[18] T. Zhao, M. Hall, H. Johansen, and S. Williams, "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*, 2021, pp. 304–317.

[19] X. You, H. Yang, Z. Jiang, Z. Luan, and D. Qian, "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, 2021, pp. 63–70.

[20] S. Zheng, R. Chen, A. Wei, Y. Jin, Q. Han, L. Lu, B. Wu, X. Li, S. Yan, and Y. Liang, "Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 874–887.

[21] X. Liu, Y. Liu, H. Yang, J. Liao, M. Li, Z. Luan, and D. Qian, "Toward accelerated stencil computation by adapting tensor core unit on gpu," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–12.

[22] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *Acm Sigplan Notices*, vol. 35, no. 5, pp. 145–156, 2000.

[23] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *SC'00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE, 2000, pp. 32–32.

[24] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 13–24.

[25] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," *ACM sigplan notices*, vol. 42, no. 6, pp. 235–244, 2007.

[26] P. Basu, M. Hall, S. Williams, B. Van Straalen, L. Oliker, and P. Colella, "Compiler-directed transformation for higher-order stencils," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 313–323.

[27] S. P. E. Corporation, "Spec2000," accessed: 2024-03-26.

[28] R. Courant and D. Hilbert, *Methods of mathematical physics: partial differential equations*. John Wiley & Sons, 2008.

[29] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[30] C. Yount, J. Tobin, A. Breuer, and A. Duran, "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)*. IEEE, 2016, pp. 30–39.

[31] G. W. Bluman and J. D. Cole, "The general similarity solution of the heat equation," *Journal of mathematics and mechanics*, vol. 18, no. 11, pp. 1025–1042, 1969.

[32] J. F. Epperson, *An introduction to numerical methods and analysis*. John Wiley & Sons, 2013.

[33] C. Andreolli, P. Thierry, L. Borges, G. Skinner, C. Yount, J. Jeffers, and J. Reinders, "Characterization and optimization methodology applied to stencil computations," *High Performance Parallelism Pearls*, pp. 377–396, 2015.

[34] K. Li, L. Yuan, Y. Zhang, Y. Yue, and H. Cao, "An efficient vectorization scheme for stencil computation," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 650–660.

[35] Nvidia, "Parallel thread execution isa version 8.4," https://docs.nvidia.com/cuda/parallel-thread-execution, accessed: 2024-03-26.

[36] ——, "Nvidia nsight compute," https://developer.nvidia.com/nsight-compute, accessed: 2024-03-26.

[37] D. Nuzman and A. Zaks, "Outer-loop vectorization: revisited for short simd architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 2–11.

[38] L. Yuan, H. Cao, Y. Zhang, K. Li, P. Lu, and Y. Yue, "Temporal vectorization for stencils," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[39] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "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, 2011, pp. 225–245.

[40] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 65–76.

[41] P. S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L.-N. Pouchet, and P. Sadayappan, "Associative instruction reordering to alleviate register pressure," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 590–602.

[42] R. De La Cruz and M. Araya-Polo, "Algorithm 942: semi-stencil," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 3, pp. 1–39, 2014.

[43] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 1991, pp. 30–44.

[44] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, 1989, pp. 655–664.

[45] T. M. Malas, G. Hager, H. Ltaief, and D. E. Keyes, "Multidimensional intratile parallelization for memory-starved stencil computations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 3, pp. 1–32, 2017.

[46] C. Ding and Y. He, "A ghost cell expansion method

for reducing communications in solving pde problems," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001, pp. 50–50.

[47] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "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*. IEEE, 2010, pp. 1–13.

[48] G. Jin, J. Mellor-Crummey, and R. Fowler, "Increasing temporal locality with skewing and recursive blocking," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001, pp. 43–43.

[49] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 215–228, 1999.

[50] D. Wonnacott, "Achieving scalable locality with time skewing," *International Journal of Parallel Programming*, vol. 30, pp. 181–221, 2002.

[51] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.

[52] U. Bondhugula, V. Bandishti, and I. Pananilath, "Diamond tiling: Tiling techniques to maximize parallelism for stencil computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1285–1298, 2016.

[53] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache oblivious parallelograms in iterative stencil computations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 49–59.

[54] L. Yuan, Y. Zhang, P. Guo, and S. Huang, "Tessellating stencils," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[55] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[56] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[57] S. Liu, Z. Zhang, and W. Wu, "Dhts: A dynamic hybrid tiling strategy for optimizing stencil computation on gpus," *IEEE Transactions on Computers*, 2023.

[58] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "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*, 2020, pp. 199–211.

[59] T. L. Falch and A. C. Elster, "Register caching for stencil computations on gpus," in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2014, pp. 479–486.

[60] N. Maruyama and T. Aoki, "Optimizing stencil computations for nvidia kepler gpus," in *Proceedings of the 1st international workshop on high-performance stencil computations, Vienna*. Citeseer, 2014, pp. 89–95.

[61] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for gpus: automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, 2013, pp. 24–31.

[62] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 311–320.

[63] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.

[64] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "On optimizing complex stencils on gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 641–652.

[65] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-specific multi-level ir rewriting for gpu: the open earth compiler for gpu-accelerated climate simulation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–23, 2021.

[66] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.