ConvStencil: Transform Stencil Computation to Matrix Multiplication on Tensor Cores

Yuetao Chen*
Microsoft Research
Beijing, China

Kun Li†
Microsoft Research
Beijing, China

Yuhao Wang*
Microsoft Research
University of Science and Technology of China
Hefei, China

Donglin Bai
Microsoft Research
Beijing, China

Lei Wang*
Microsoft Research
University of Chinese Academy of Sciences
Beijing, China

Lingxiao Ma
Microsoft Research
Beijing, China

Liang Yuan
Chinese Academy of Sciences
Beijing, China

Yunquan Zhang
Chinese Academy of Sciences
Beijing, China

Ting Cao
Microsoft Research
Beijing, China

Mao Yang
Microsoft Research
Beijing, China

Abstract

Tensor Core Unit (TCU) is increasingly integrated into modern high-performance processors to enhance matrix multiplication performance. However, constrained to its over-specification, its potential for improving other critical scientific operations like stencil computations remains untapped.

This paper presents ConvStencil[1], a novel stencil computing system designed to efficiently transform stencil computation to matrix multiplication on Tensor Cores. We first develop a performance model for ConvStencil to guide algorithm design and optimization on TCUs. Based on this model, we propose three techniques: (1) Memory-efficient Layout Transformation using the stencil2row method; (2) Computation-dense Compute Adaptation with Dual Tessellation and kernel fusion; and (3) Performance-boosting Conflict Removal using a Lookup Table and Dirty Bits Padding. ConvStencil outperforms other stencil optimization frameworks, achieving significant speedups compared to solutions like AMOS, cuDNN, Brick, DRStencil, and TCStencil. By transforming stencil computation on Tensor Cores, ConvStencil promises to improve the performance of various scientific and engineering applications.

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

Keywords: Stencil Computation, Convolution, Matrix Multiplication, Tensor Cores

ACM Reference Format:
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 *The 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24), March 2–6, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627535.3638476

1 Introduction

As deep learning models become more prevalent, primarily characterized by matrix multiplication (MM) operations, processors both existing and emerging have increasingly incorporated specialized units to expedite MM. These specialized units are known as Tensor Core Units (TCUs) which

*Work done during an internship at Microsoft Research.
†Corresponding author (kunli@microsoft.com).
[1]ConvStencil is available at https://github.com/microsoft/ConvStencil.

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03…$15.00
https://doi.org/10.1145/3627535.3638476

provide substantial performance acceleration for MM-based deep learning models [12], such as Tensor Cores in NVIDIA GPUs.

While Tensor Cores could deliver a promising performance, it is essential to note that the computing patterns in HPC field are considerably more diverse and complicated. Most of them are hard to be directly expressed with MM. Stencil, identified as one of seven performance-critical computing patterns by Berkeley view, is a representative one of them [3, 4, 23].

A stencil contains a pre-defined pattern that updates each point in d-dimensional spatial grid iteratively along the time dimension. The value of one point at time t is a weighted sum of itself and neighboring points at the previous time $t-1$. Stencil serves as one of the most important kernels widely used in science and engineering, such as fluid dynamics [20, 25], earth modeling [21], and weather simulations [2, 6].

Currently, a limited number of studies have explored Tensor Cores for non-MM operations. Initial work has implemented simple reduction and scan primitives on Tensor Cores, marking the first attempts to expand the range of non-MM operations that can be expressed as Tensor Core operations [13]. More recent research, TCStencil, has sought to apply Tensor Cores to more complex computation patterns like stencil [24]. However, TCStencil suffers from poor algorithmic generality and low Tensor Core utilization. On one hand, TCStencil is constrained to symmetric MM on FP16 Tensor Cores (i.e. matrix multiplication of matrices with the same shape), while most stencil computation necessitates FP64 precision and only specific asymmetric MM is supported on FP64 Tensor Cores. On the other hand, TCStencil encounters uncoalesced access to global memory and bank conflicts within shared memory, preventing the computing power of Tensor Cores from being fully exploited. To the best of our knowledge, there is no other work that provides a practical way to adapt stencil computation on Tensor Cores effectively.

This paper presents a novel stencil computing system, ***ConvStencil***, designed to transform stencil computation to matrix multiplication on Tensor Cores efficiently.

The design of ConvStencil is based on a crucial observation that stencil in high-performance computing and convolution in deep learning exhibit similarities in their computational patterns. Both approaches involve the use of a stencil kernel (or convolution kernel) to form a sliding window, performing weighted computations on the data within the window on the input matrix. To efficiently support convolution on Tensor Cores, im2row(col) method is used in GEMM-based convolution computations [9]. It involves converting the input and filter into matrices, allowing convolution to be computed by MM.

Guided by this observation, the key insight of ConvStencil is inspired: since the computation patterns of stencil and convolution are so similar, why not build a bridge between stencil computation and Tensor Core using the im2row mechanism? However, given the critical differences in algorithmic details between stencil and convolution, this is still not a low-hanging fruit as several considerable technical challenges must be tackled.

Firstly, the application of im2row to convolution operations enables their transformation into MM. However, this transformation results in matrix-vector multiplication due to the fact that both the number of stencil kernels and the number of channels are one during each iteration, potentially causing significant memory expansion and low Tensor Core utilization. Secondly, the FP64 Tensor Core operations exclusively support a unique asymmetric small MM, which presents challenges for efficient algorithm adaptation under this constraint. Moreover, the algorithm's implementation and design may encounter performance-affecting conflicts between algorithm implementation and hardware design, such as warp divergence and bank conflicts, leading to a substantial decline in performance.

The ConvStencil consists of three key techniques to address the aforementioned challenges: memory-efficient *Layout Transformation*, computation-dense *Compute Adaptation*, and performance-boosting *Conflicts Removal*.

In Layout Transformation, we introduce stencil2row to create an efficient memory layout for MM with reduced memory consumption. It achieves a 70.0% to 96.4% memory footprint reduction compared to im2row. In Compute Adaptation, we propose Dual Tessellation to enhance Tensor Core utilization through matrix tessellation, increasing Tensor Core utilization from 12.5% to 87.5%. Concurrently, Kernel Fusion reduces matrix sparsity to further improve computational density on Tensor Cores. In Conflicts Removal, we design a Lookup Table to avoid costly operations and reduce redundant addressing calculations. Moreover, Dirty Bits Padding uses a padding zone to write dirty data and evade conditional branches, thus achieving a conflict-free implementation for further boosting performance. In comparison to TCStencil which also utilizes Tensor Cores, ConvStencil reduces the non-coalesced global memory accesses by 44.0% and the bank conflicts per request by 63.5%, on average.

Results are demonstrated from three aspects by using a diverse set of stencil kernels. First, our designs and optimizations prove to be effective, with each proposed technique contributing to a measurable performance improvement. Second, ConvStencil outperforms five state-of-the-arts (cuDNN [11, 31], AMOS [52], Brick [49–51], DRStencil [43] and TCStencil [24]) in various benchmarks. Third, ConvStencil is also superior to DRStencil with three-time-step fusion, showing that our performance gains are not only from kernel fusion optimization but also from our algorithmic design.

Our contributions are highlighted as follows.

- We propose ConvStencil, a novel stencil computing system designed to transform stencil computation to matrix multiplication on Tensor Cores efficiently.
- We propose Stenci2row layout transformation. It reduces the redundancy in the im2row result and remains an efficient memory layout for MM operations.
- Compute Adaptation adopts Dual Tessellation to enhance Tensor Core utilization and Kernel Fusion to further improve computational density on Tensor Cores.
- Conflicts Removal presents Lookup Table and Dirty Bits Padding to eliminate performance-affecting conflicts for further performance improvements.

2 Background and Challenges

2.1 Stencil Computation

Stencil computation, a widely adopted technique in scientific and engineering domains, involves the iterative updating of multi-dimensional inputs according to a predefined computation pattern. This predefined pattern, referred to as the *shape*, primarily consists of two types: star and box. A star stencil computes the weighted sum of a central point and its neighboring points, which diverge from the central point in a single dimension only. A box stencil calculates the weighted sum of a square or cube, wherein the central point is located at the core of the geometric shape. The extent of points involved in a specific computation pattern is dictated by the *radius*, also referred to as *order*. For instance, the computation pattern for a box stencil with a radius of 1 constitutes a 3×3 square.

2.2 GEMM-based Convolution on Tensor Cores

Tensor cores represent a specialized hardware component, developed by NVIDIA, designed to accelerate matrix multiplications. Its unique capacity to perform mixed-precision matrix multiplication and accumulation (MMA, as demonstrated in Equation 1), allows for processing speeds superior to those of CUDA cores.

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

The GEMM-based convolution converts convolution into MM and becomes an efficient method for computing convolution on Tensor Cores. The procedure of GEMM-based convolution is shown in Figure 1. In the procedure, the multi-channel input and the convolutional kernels are both reshaped into 2D matrices and then the convolution operation is expressed as a MM. The input matrix is created by unrolling each kernel-sized patch of the image into a row (im2row). The kernel (or filter) matrix is created by unrolling the filter weights into a column. The convolution operation comprises multiple convolution kernels, typically in powers of 2. Columns reshaped from convolutions form the kernel matrix. The MM operation is then applied to these two matrices.

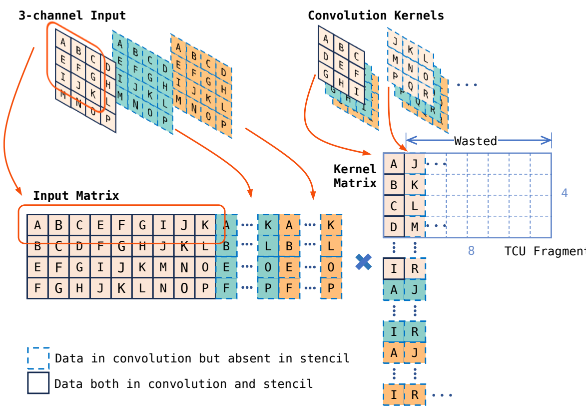

Figure 1. GEMM-based convolution and stencil.

2.3 Challenges

Convolution and stencil computations share a high degree of similarity. They both slide the kernel over input grids and compute the weighted sum. Despite extensive research, there remains a lack of effective and practical methods for efficiently utilizing Tensor Cores in stencil computations. This leads to the question of why stencil computations struggle to be mapped to Tensor Cores as conveniently as convolutions. Here we identify and discuss three primary challenges that contribute to this issue.

1. Space explosion. Adopting the im2row transformation to convert stencil to MM is a straightforward idea. However, the im2row transformation demands high memory requirements, with the memory footprint of the resulting matrix several times or even dozens of times larger than the original input, leading to space explosion. For example, for a 10×10 input and a 3×3 kernel, the size of the input matrix is expanded to 100×9, which is 9x larger than the original input. For common convolutions, space explosion will not become a concerning issue because enough columns of the kernel matrix densify the matrix multiplication, which achieves a balance between memory and computation overheads. However, after the im2row transformation, as illustrated in Figure 1, stencil computation is converted into a matrix-vector multiplication. Due to the sparsity of matrix-vector multiplication in Tensor Cores, the space explosion of im2row becomes concerning. Furthermore, stencil computations usually require FP64 precision, further exacerbating memory demands. In sharp contrast to this, the shared memory available on a GPU is limited; even on an A100, each Streaming Multiprocessor (SM) has only 164KB of shared memory [30].

2. Low Tensor Core utilization. As shown in Figure 1, the convolution is converted into stencil computation when these two requirements are satisfied. 1) The channel of input data and convolution kernels is 1. 2) Stencil computation only exists one kernel. At this point, the stencil computation becomes a matrix-vector multiplication. However, for

FP64 precision, Tensor Cores on NVIDIA A100 only support $8 \times 8 \times 4$ MMA ($m = 8$, $n = 8$ and $k = 4$ in Equation 1), which means 7/8 columns of the matrix being multiplied on the right are wasted.

3. Conflicts in algorithm and hardware. Upon completing the design of the algorithm for the Tensor Core, it becomes evident that there are two significant conflicts between the algorithm implementation and the hardware design during the mapping process. 1) A significant number of repetitive offset calculations for memory access arise, leading to conflicts with standard stencil computations. These conflicts consume computational resources and result in performance degradation. 2) A multitude of conditional branches and bank conflicts exist in layout transformation, leading to severe warp divergence and serial memory access.

3 ConvStencil

ConvStencil represents a novel approach to stencil computation, leveraging Tensor Cores via convolution-like methodologies. We first introduce our theoretical performance model. Then we introduce the fundamental components of ConvStencil, including layout transformation, compute adaptation, and conflict removal.

During the layout transformation phase, we propose *stencil2row* that reshapes the input into two distinct and smaller matrices, primed for subsequent Tensor Core computations. In the compute adaptation phase, dual tessellation iteratively applies Tensor Core MMA on tiles selected from the stencil2row matrix to generate the stencil results. For the conflicts removal part, we precompute pointer offsets to prevent time-consuming integer division and modulus operations. We also propose dirty bits padding that removes load bank conflicts and eliminates conditional branches via utilizing padding area.

3.1 Performance Model

In order to demonstrate the performance improvement of ConvStencil theoretically, we build the performance model, which is shown in Equation 2, 3 and 4:

$$\mathcal{T} = max(\mathcal{T}_{compute}, \mathcal{T}_{memory}) \tag{2}$$

$$\mathcal{T}_{compute} = \frac{1}{fN_{tcu}} \sum_{i=0}^{K_{tcu}} (k_{tcu_i} \times CPI_{tcu_i}) \tag{3}$$

$$\mathcal{T}_{memory} = max(\frac{data_R}{bw_G} + \frac{data_W}{bw_G}, \frac{data_{transW}}{bw_S} + \frac{data_{transR}}{bw_S}) \tag{4}$$

The explanation of used symbols is listed in Table 1. The compute time and memory access time constitute the overall time of stencil computations.

The time required for computation is the product of the inverse of the clock frequency and the number of cycles required. The number of cycles required is computed by summing the products of the number of each type of instruction in the program and the number of cycles that instruction takes. On NVIDIA A100 GPU, the number of cycles of an FP64 MMA instruction on TCU is 16 [1]. The time required for memory access is the maximum sum of read/write time across different memory hierarchies. Through this theoretical performance model, we analyze the performance advantages of ConvStencil in Section 3.3.

3.2 Layout Transformation

Stencil2row. Current im2row transformation suffers memory explosion. When the original input is transformed into an im2row matrix, the demand for memory inflates by several times. Figure 2 demonstrates this phenomenon using a 7×7 convolution kernel as an example. Upon transforming a $m \times n$ input via im2row, an $(m-6)(n-6) \times 49$ im2row matrix is formed. As the kernel size increases, the memory required for the im2row transformation escalates.

The stencil2row is proposed based on the following three observations. 1) When the original input is transformed into an im2row matrix, most elements in the im2row matrix are redundant and the transformation causes a space explosion. As shown in Figure 2, the elements of its $1^{st} \sim 6^{th}$ rows are all repetitions of the elements in the 0^{th} and 7^{th} rows. 2) In the im2row transformation, we observe that the data sequencing in redundant rows has been already stored beyond the redundant rows. For example, the 3^{rd} row of im2row matrix in Figure 2 can be divided into two parts (sandy brown and light blue). The data sequencing of the first part (sandy brown) can be found in the 0^{th} row, while the data sequencing of the second part (light blue) can be found in the 7^{th} row. This observation suggests that the structure of intermediate rows (e.g. $1^{st} \sim 6^{th}$ rows) containing redundant data is subsumed by other rows (e.g. 0^{th} and 7^{th} rows), indicating that there exists the potential to construct the outcome of

Table 1. Notations.

Symbols	Meanings
$\mathcal{T}$	Overall core time
$\mathcal{T}_{compute}$	Core time of computing
$\mathcal{T}_{memory}$	Core time of memory transactions
f	GPU frequency (core clock)
N_{tcu}	Number of TCUs
K_{tcu}	Types of TCU instructions
k_{tcu_i}	Number of the i^{th} type of TCU instructions
CPI_{tcu_i}	Cycles per the i^{th} type of TCU instruction
$data_R$	Amount of data read from GM[1]
$data_W$	Amount of data written to GM
$data_{transW}$	Amount of transformed data written to SM[2]
$data_{transR}$	Amount of transformed data read from SM
bw_G	Bandwidth of global memory
bw_S	Bandwidth of shared memory

[1] GM denotes global memory.
[2] SM denotes shared memory.

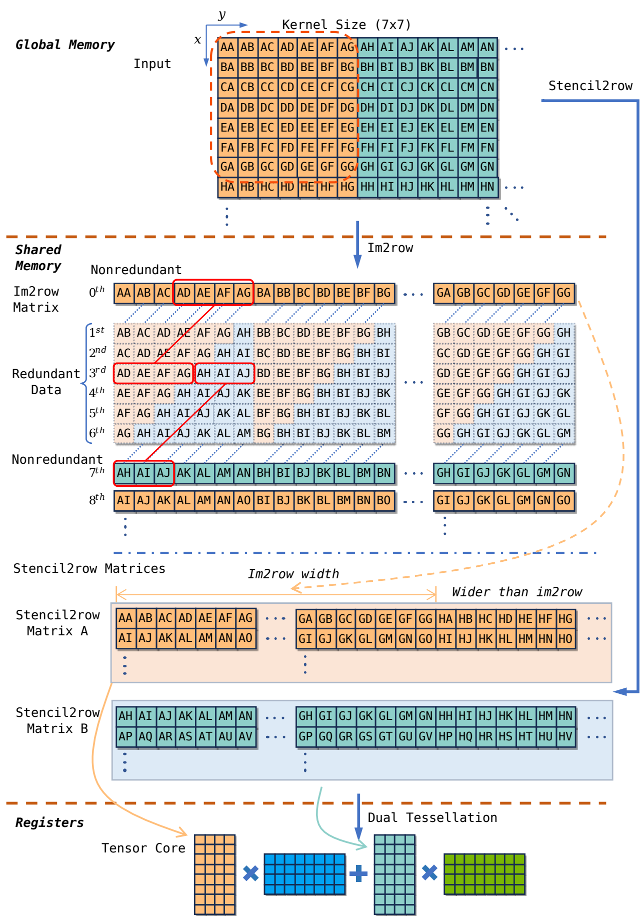

Figure 2. Stencil2row and its comparison with im2row.

intermediate rows solely with other rows (e.g. 0^{th} and 7^{th} rows). 3) Shared memory resides on-chip, so it has much lower latency than global memory. Table 2 shows the access latencies of different memory types [1]. The access latency of global memory exceeds that of shared memory by more than an order of magnitude.

Based on these three observations, we propose *stencil2row*. Stencil2row transforms the original input into two smaller matrices. In Figure 2, these two matrices are marked as *Stencil2row Matrix A & B*. The 0^{th} row of the stencil2row matrix A can be viewed as an extension of the 0^{th} row of the im2row matrix. The 0^{th} row of stencil2row matrix A extends to the last row of the original input matrix. In other words, the final elements of the stencil2row matrix A are the elements of the last row of the original input matrix. Next, the 1^{st} row of stencil2row matrix A can be viewed as an extension of the 8^{th} row of the im2row matrix. This pattern continues in the same manner and the stencil2row matrix A is constructed. The mapping function of stencil2row matrix A is written as a vector function in Equation 5,

Table 2. Memory access latencies [1].

Memory access types	Cycles
Global memory	290
Shared memory (load/store)	23/19

$$\mathbf{Y} = stencil2row_A(\mathbf{X}) = \begin{bmatrix} \lfloor y/(n_{kernel}+1) \rfloor \\ n_{kernel}x + y \bmod (n_{kernel}+1) \end{bmatrix} \tag{5}$$

where

$$\mathbf{X} = \begin{bmatrix} x \\ y \end{bmatrix}, (y+1) \bmod (n_{kernel}+1) \neq 0$$

$\mathbf{X}$ indicates the index of the original input elements. $\mathbf{Y}$ indicates the index of stencil2row matrix A elements. n_{kernel} is the edge length of the kernel. The construction of stencil2row matrix B is similar, which is shown in Equation 6,

$$\mathbf{Y} = stencil2row_B(\mathbf{X}) = \begin{bmatrix} \lfloor (y-n_{kernel})/(n_{kernel}+1) \rfloor \\ n_{kernel}x + (y-n_{kernel}) \bmod (n_{kernel}+1) \end{bmatrix} \tag{6}$$

where

$$\mathbf{X} = \begin{bmatrix} x \\ y \end{bmatrix}, ((y-n_{kernel}+1) \bmod (n_{kernel}+1) \neq 0$$

After defining how stencil2row matrices form, we implicitly construct stencil2row matrices in shared memory based on the observation of different access latencies between global memory and shared memory. We construct the tiles of stencil2row matrices on the fly as original input data are loaded. Specifically, in the context of NVIDIA GPUs, we retrieve original data from global memory, subsequently construct the tiles of stencil2row matrices within shared memory, and utilize Tensor Cores to read from shared memory for matrix computations. Throughout the entire process, the stencil2row matrices are not explicitly fully constructed.

Stencil2row eliminates most redundant elements in the im2row matrix and alleviates memory pressure. Furthermore, not only does stencil2row preserve the beneficial characteristics of im2row that allow the use of matrix multiplication, but it is also more suited to the Tensor Cores specifically for stencil computations. Moreover, we construct the tiles of stencil2row matrices in shared memory on the fly as original input data are loaded, which reduces global memory load and store operations. After stencil2row transformation, matrices are computed by Tensor Core via *dual tessellation* that is introduced in Section 3.3. With the details of stencil2row described, we quantitatively analyze the advantages of stencil2row from the perspectives of memory saving and data transfer saving.

Memory Saving. For stencil2row data layout, the original input is transformed into two matrices. The numbers of rows and columns of each matrix are calculated by Equation 7 and 8,

$$m_{stencil2row} = \frac{n}{n_{kernel}+1} \tag{7}$$

$$n_{stencil2row} = n_{kernel} \times m \tag{8}$$

where m and n denote the dimensions of the input. However, for im2row data layout, the numbers of rows and columns of im2row matrix are shown in Equation 9 and 10.

$$m_{im2row} = mn \tag{9}$$

$$n_{im2row} = n_{kernel}^2 \tag{10}$$

Thus, the ratio of memory space occupied by stencil2row and im2row is defined by Equation 11.

$$\frac{stencil2row}{im2row} = \frac{2}{(n_{kernel}+1)n_{kernel}} \tag{11}$$

Table 3 shows the multiplication factors of memory expansion for im2row and stencil2row compared to the input memory under various shapes, and the amount of memory reduced in stencil2row compared to im2row. Compared to im2row, stencil2row reduces memory usage by over 70% across all shapes.

Data Transfer Saving. Though stencil2row reduces over 70% memory expansion compared to im2row, the transfer of this data still constitutes a considerable expense. Data transfer between global memory and shared memory/registers is expensive. Stencil2row saves data transfers in two aspects.

First, stencil2row implicitly constructs the tiles of stencil2row matrices in shared memory. ConvStencil only conducts a single global memory read-and-write operation, thereby not increasing the overhead of global memory read-and-write operations.

Second, compared to im2row, stencil2row reduces the occupancy of memory space, leading to a decrease in the amount of data written to shared memory. It is often difficult to eliminate store bank conflicts in shared memory, so the reduction of data written to shared memory by stencil2row is more beneficial to performance enhancement.

3.3 Compute Adaptation

After layout transformation, the question then becomes how to efficiently compute the stencil results on stencil2row matrices with Tensor Cores. To address this, we propose *dual tessellation* to efficiently exploit Tensor Cores for stencil computation. We also leverage kernel fusion to further enhance Tensor Core utilization.

Table 3. Multiplication factors of memory expansion compared to the original input.

Shapes	im2row	stencil2row	Memory saving
Heat-2D	5	1.5	70.00%
Box-2D9P	9	1.5	83.33%
Star-2D9P	9	1.67	81.49%
Box-2D25P	25	1.67	93.33%
Star-2D13P	13	1.75	86.54%
Box-2D49P	49	1.75	96.43%

Dual Tessellation. Applying the existing GEMM-based convolution methodology to stencil computation can result in poor Tensor Core utilization and memory explosion, which has already been discussed in Section 2.3 and Section 3.2 respectively. Stencil2row transformation reduces memory demand, then we need to efficiently use Tensor Cores for stencil computation based on the stencil2row matrices.

We observe that the sequencings of the redundant rows in im2row matrix, as shown in Figure 2, have been stored in nonredundant rows. Moreover, this redundancy exhibits a well-defined pattern. In Figure 2, it is demonstrated that redundant rows may be composed of multiple triangles. Each element within the brown triangles is incorporated in the first nonredundant row, while every element in the blue triangles is included in the second nonredundant row. These observations enable us to construct a highly efficient stencil algorithm on the Tensor Cores, based on the stencil2row matrices.

We propose *dual tessellation*, a novel algorithm for stencil computation based on stencil2row transformation. Dual tessellations are iteratively called to progressively compute all stencils. Each dual tessellation firstly builds two half-result matrices called *vitrolite A & B*[2]. Then, summing two pieces of vitrolite yields the stencil computation result, which is termed tessellation.

In Figure 3, dual tessellation encompasses three steps.

In Step 1, a tile[3] from stencil2row matrix A needs to be multiplied by weight matrix A to build vitrolite A. We introduce the tile and weight matrix A respectively.

The dual tessellation process iteratively retrieves a tile from the stencil2row matrix A. This tile comprises 8 rows, because of the fact that the number of rows in the matrix being left-multiplied by the Tensor Core is 8. The column number of the tile is n_{kernel}^2. As exemplified by Box-2D49P in Figure 3, the size of a tile is 8×49. Each dual tessellation retrieves a different tile from the stencil2row matrix A. Equation 12 presents the base address of each tile,

$$base\ address_i = 8n_{stencil2row}\left\lfloor\frac{i}{m}\right\rfloor + (i\ mod\ m)n_{kernel} \tag{12}$$

where $i \in \{0, 1, 2, \cdots\}$. Intuitively, it means that each tile shifts n_{kernel} elements to the right after dual tessellation. Once the first eight rows are computed, the next eight rows are processed until the end of stencil2row matrix A.

The size of weight matrix A is $n_{kernel}^2 \times n_{kernel}$. In Figure 3, the size of the weight matrix is 49×7 and is padded to 49×8 for the Tensor Core MMA operations. The weight matrix A is composed of seven lower triangular matrices concatenated together. The first column of weight matrix A contains all the 49 weights ($a_1 \sim a_{49}$), so the product of the tile from stencil2row matrix A and the first column computes 8 complete

[2]Vitrolite is a kind of pigmented glass with different colors and was often tessellated on walls for decoration in the 20th century.

[3]Here, a tile refers to a portion of a matrix.

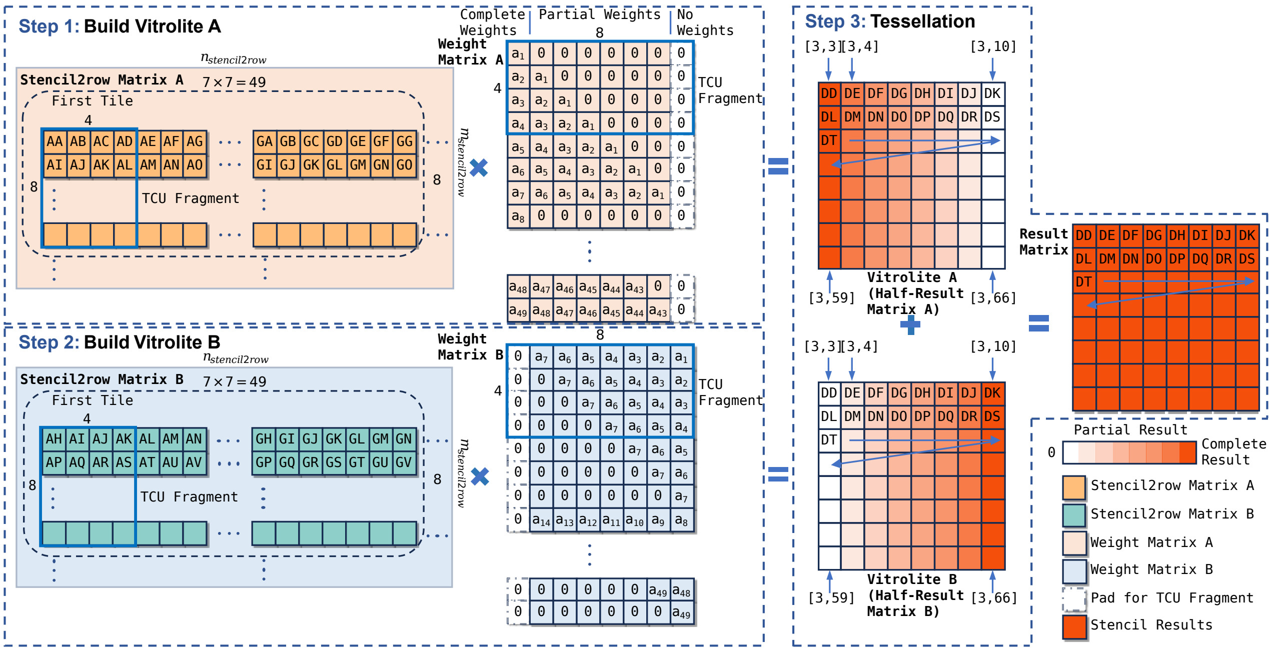

Figure 3. Dual tessellation of stencil computation.

stencil results. In Figure 3, this product is the first column of vitrolite A (half-result matrix A) as indicated in the darkest red. The second to seventh columns of weight matrix A contain partial weights; hence, the second to seventh columns of vitrolite A constitute partial stencil computation results. The gradation of red in Figure 3 indicates the proportion of the stencil computation accomplished. The last column of weight matrix A is entirely zeros, which in turn results in the last column of vitrolite A also being composed of zeros, as indicated in white. At this point, we have built vitrolite A and completed Step 1.

Step 2 is similar to Step 1, but it retrieves tiles from stencil2row matrix B and uses a different weight matrix B. Weight matrix B is composed of upper triangular matrices. The purpose of this design is to align the two product matrices so that they can be directly added together. Vitrolite B is the product of a tile from stencil2row matrix B and weight matrix B. Under meticulous design, vitrolite B is the opposite: the first column is entirely composed of zeros, while the last column contains complete stencil computation results, each position corresponding directly to that of vitrolite A.

In Step 3, called tessellation, by summing vitrolite A and vitrolite B, we obtain the result of the stencil computation. As exemplified by Box-2D49P in Figure 3, the index of the first dual tessellation results is [3][3:66]. Finally, we write back the results to global memory.

Since the Tensor Core MMA operation can fuse matrix multiplication and accumulation, we did not calculate vitrolites A & B separately and then add them together in the

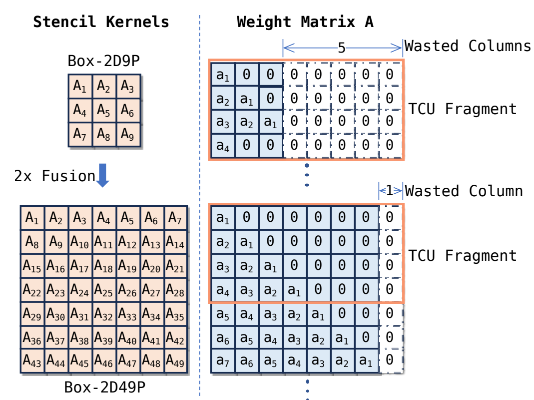

Figure 4. Kernel fusion to increase Tensor Core utilization.

implementation. Instead, after calculating vitrolite A, the results of each matrix multiplication in the calculation of vitrolite B are accumulated on vitrolite A. This approach reduces one MMA operation for each dual tessellation. The number of MMA operations in a dual tessellation is $2\lceil n_{kernel}^2/4\rceil$.

For stencil computation, dual tessellation significantly improves Tensor Core utilization and is compatible with our stencil2row transformation.

Kernel Fusion Dual tessellation can be applied to any stencil kernel. Nevertheless, some small kernels struggle to efficiently utilize the Tensor Cores. Therefore, we temporally fuse some stencil kernels for densifying the computations in Tensor Cores. For example, in Figure 4, weight matrix A of Box-2D9P has only 3 columns, which wastes 5 columns in

Tensor Core fragments. To enhance the utilization of Tensor Cores, we performed two temporal fusions, converting Box-2D9P into Box-2D49P. After kernel fusions, only 1 column of Tensor Core fragments is wasted, thereby improving the utilization of Tensor Cores.

Quantitative Performance Analysis For a better understanding of the advantages of ConvStencil compared to convolution for stencil computations, we conduct a quantitative analysis of ConvStencil's performance.

We analyze the performance of our ConvStencil and GEMM-based convolution based on the theoretical performance model discussed in Section 3.1. According to Equation 2, since the total time is the maximum of computation time and memory access time, we analyze computation time and memory access time separately.

Computation time analysis. Each dual tessellation compute stencils. Thus, for the $m \times n$ input, the number of required dual tessellations is $mn/(8n_{kernel}+8)$. Because the number of MMA operations in a dual tessellation is $2\lceil n_{kernel}^2/4\rceil$, the number of MMAs required by ConvStencil is shown in Equation 13.

$$N_{MMA} = \frac{2mn}{8 \times (n_{kernel}+1)} \left\lceil \frac{n_{kernel}^2}{4} \right\rceil \tag{13}$$

Thus, the computation time of ConvStencil is shown in Equation 14,

$$\mathcal{T}_{compute_{ConvStencil}} = \frac{\frac{2mn}{8(n_{kernel}+1)} \left\lceil \frac{n_{kernel}^2}{4} \right\rceil \times CPI_{tcu}}{f N_{tcu}} \tag{14}$$

where, in the A100 FP64 context, f is 1410 MHz, N_{tcu} is 432 and CPI_{tcu} is 16 cycles [1, 32].

However, the computation time of using GEMM-based convolution to compute stencil is shown in Equation 15.

$$\mathcal{T}_{compute_{GEMM-basedConv}} = \frac{\frac{n_{kernel}^2 mn}{32} \times CPI_{tcu}}{f N_{tcu}} \tag{15}$$

Due to the orders of stencils is always greater than one, $n_{kernel} \geq 3$. Thus, the computation time of ConvStencil is less than that of GEMM-based convolution.

Memory access time analysis. We assume that the implementation of GEMM-base convolution is implicit that will not introduce overhead of loading or storing data in global memory. Thus, based on Equation 4, $data_R$, $data_W$, bw_G, and bw_S are constants. We only need to analyze $data_{transW}$ and $data_{transR}$. As shown in Equation 11, $data_{transW}$ of ConvStencil is only $2/((n_{kernel}+1)n_{kernel})$ of that of GEMM-based convolution. $data_{transR}$ of ConvStencil is $2/(n_{kernel}+1)$ of that of GEMM-based convolution. Therefore, the memory access time of ConvStencil is less than that of GEMM-based convolution.

As both the computation time and memory access time for ConvStencil are less than those for GEMM-based convolution, ConvStencil outperforms GEMM-based convolution in terms of stencil computations.

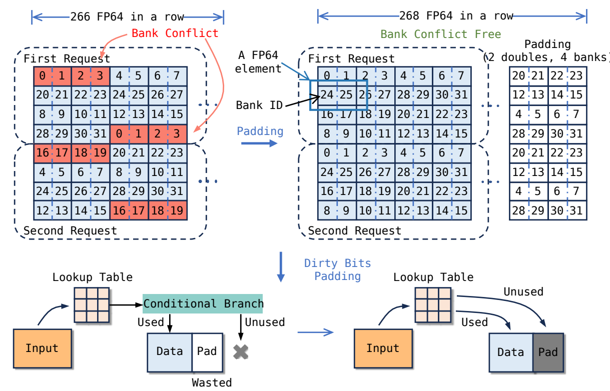

Figure 5. Dirty bits padding eliminates load bank conflicts and conditional branches.

3.4 Conflicts Removal

After introducing layout transformation and compute adaptation, three conflicts hidden in ConvStencil undermine the performance. 1) A large number of integer division and modulus operations are inevitably involved for indexing in layout transformation. This causes a conflict between the introduced computation interrupts and continuous data transfers.2) Bank conflicts that occur during dual tessellations limit the shared memory bandwidth. 3) Because stencil2row matrix A or B is smaller than the original input, conditional statements are applied to determine whether the data is required or not. These conditional branches introduce conflicts in thread control. To remove these three conflicts, we introduce lookup tables and dirty bits padding.

Lookup Table. In the process of layout transformation, the address pointer offsets need to be computed for transforming data from global memory to shared memory. These computations contain a large number of integer division and modulus operations that are highly time-consuming on GPUs. Moreover, these offset computations are redundant across different blocks. To reduce computational overhead in the layout transformation process, we precompute the pointer offsets in the host and provide them to the CUDA kernel as lookup tables.

Dirty Bits Padding. The padding area used to alleviate bank conflicts is filled with dirty data to eliminate conditional branch statements. In dual tessellations, bank conflicts usually occur when Tensor Cores load data from stencil2row matrices in shared memory. The bank conflict arises when multiple threads within a single warp simultaneously access different addresses of the same bank. The hardware splits this request into multiple independent conflict-free requests, which diminishes the shared memory throughput.

We use paddings to remove load bank conflicts in shared memory. The padding adds extra space to change the way data is mapped into shared memory. Figure 5 exemplifies, with the case of the stencil2row matrix A (with 266 columns), why padding removes load bank conflicts. On A100 GPU, the bank size is 4 bytes, which means the 1 FP64 element occupies 2 banks. In CUDA WMMA API, a warp (32 threads) loads a 8×4 matrix fragment, so each thread reads one FP64 type. However, 32 FP64 elements occupy 64 banks, and a warp read from up to 32 different banks at one time. Thus, a 8×4 matrix fragment read is composed of two shared memory requests. The first 16 threads read the 4×4 fragment at the front, followed by the last 16 threads reading the 4×4 fragment at the back. Thus, the unit to check for bank conflicts should be a 4×4 fragment. In Figure 5, without padding, A[0][0:3] and A[3][4:7] both fall in bank 0-3, resulting in bank conflicts of the first request. A similar situation applies to the second request. After padding of two FP64 elements, the first and second requests of 4×4 fragments are equally distributed in 32 different banks, leading to load bank conflict free.

However, typically padding area is wasted after changing the memory layout. We found that unused data (dirty bits) can be dumped into the padding space, which eliminates the conditional branches and corresponding computation. As introduced in Section 3.2, stencil2row transforms the original input into two matrices and the size of each matrix is smaller than the original matrix. This suggests that, for each transformed matrix, some elements of the input cannot be mapped into the transformed matrix, which introduces the conditional branches and corresponding comparison operations. As illustrated in Figure 5, with dirty bits padding, unused data are mapped into the padding area via the lookup table and will not be used. After this optimization, no conditional branch statements are needed to select data to be used, thus improving the performance of stencil computations.

4 Generalization

After introducing ConvStencil in 2D, ConvStencil can be easily generalized to other dimensions.

4.1 1D

For 1D stencil, the shape of stencil2row matrices changes. The number of rows and columns in a stencil2row matrix are $n/(n_{kernel}+1)$ and n_{kernel}, respectively. n_{kernel} indicates the length of the kernel and n indicates the size of inputs. After layout transformation, the 1D computation process of ConvStencil is identical to the 2D ConvStencil. The dual tessellation is iteratively applied to compute all stencils.

4.2 3D

The 3D stencil computation can be decomposed into 2D stencil computations with different weights, which are calculated using ConvStencil, and then summed over different 2D planes. In star shape of 3D stencil, each 2D plane has a different size. We use CUDA cores to compute small planes, while tensor cores are used for large planes. Although commercial GPUs do not provide an interface for warp scheduling to explicitly implement parallel computing between Tensor Cores and CUDA cores, the utilization of both Tensor Cores and CUDA cores can afford opportunities for GPU scheduling to leverage these two types of computing units parallelly [48].

5 Evaluation

5.1 Setup

Implementation. We implement ConvStencil with CUDA C++ and WMMA APIs. ConvStencil is compiled with NVCC 12.2.

Platform. Our experimental platform is composed of an AMD EPYC 7V13 processor and an NVIDIA A100 Tensor Core GPU. The A100 GPU we use is connected to the motherboard via PCIe Gen4, with a transmission bandwidth of 64GB/s. Our A100 GPU possesses 80GB of HBM2e memory with 1935GB/s memory bandwidth. The A100 GPU features 108 SMs, with each SM comprising 4 Tensor Cores. The Tensor Cores deliver a peak FP64 performance of 19.5 TFLOPS. Our platform also possesses 216GB DDR4 DRAM memory in 8 channels.

State-of-the-arts. We compare ConvStencil with a wide range of state-of-the-arts, including cuDNN [11, 31], AMOS [52], Brick [49–51], DRStencil [43], and TCStencil [24] in FP64.

We use cuDNN convolution API and set $channel = 1$ to compute stencils with FWD_IMPLICIT_PRECOMP_GEMM algorithm which is the most related to ConvStencil. AMOS supports depth-wise convolutions that are computationally equivalent to stencil operations. Because it requires space searches for better mappings, we use the results after 1,000 search trials. TCStencil is designed to support only FP16 precision for stencil computation. Due to the different matrix shapes between FP16 and FP64 on Tensor Cores, it cannot be directly converted into FP64 precision. For the same memory bandwidth, the speed of reading and writing FP16 data is four times that of FP64. Moreover, on the Tensor Cores of A100, the computation speed of FP16 is 16 times that of FP64. Therefore, if TCStencil is modified to support FP64, in the best case, its speed (GStencils/s) will be reduced to a quarter of the original. Thus, we conduct the comparison by dividing the speed of TCStencil by 4 in our evaluation.

Benchmarks. We apply various stencil kernels for benchmarks, including Heat-1D, 1D5P, Heat-2D, Box-2D9P, Star-2D13P, Box-2D49P, Heat-3D, and Box-3D27P, which is shown in Table 4 in detail [18, 45].

Table 4. Configuration of benchmarks.

Kernels	Points	Problem size	Block size
Heat-1D	3	10240000×100000	1024
1D5P	5	10240000×100000	1024
Heat-2D	5	$10240 \times 10240 \times 10240$	32×64
Box-2D9P	9	$10240 \times 10240 \times 10240$	32×64
Star-2D13P	13	$10240 \times 10240 \times 10240$	32×64
Box-2D49P	49	$10240 \times 10240 \times 10240$	32×64
Heat-3D	7	$1024 \times 1024 \times 1024$	8×64
Box-3D27P	27	$1024 \times 1024 \times 1024$	8×64

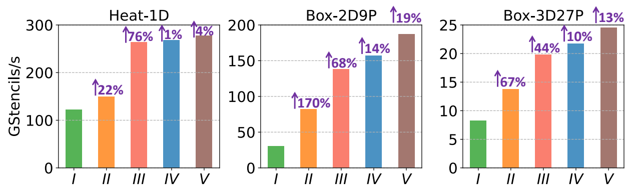

Figure 6. Performance breakdown of ConvStencil.

Metrics. Most studies on stencil [10, 28, 44–47] exhibit their results in terms of GStencils/s (GCells/s) representing how many stencil points are updated per second, which is defined in Equation 16,

$$GStencils/s = \frac{T \times \prod_{x=1}^{n} N_x}{t \times 10^9} \tag{16}$$

where T indicates the iteration round; N_x indicates the problem size of the x^{th} dimension; t indicates the execution time.

5.2 Performance Breakdown

In this subsection, we investigate how ConvStencil benefits from different optimizations. We illustrate the performance breakdown of ConvStencil on three benchmarks, including Heat-1D, Box-2D9P, and Box-3D27P, because these are representative complex shapes across different dimensions.

As can be seen from Figure 6, our stencil2row transformation provides 22%, 170%, 67% speedup compared to explicit transformation in global memory, in Heat-1D, Box-2D9P, and Box-3D27P, respectively. This performance improvement comes from reducing data transfers. Stencil2row performs read and write operations on 100% of the original data in the global memory, without introducing any additional overhead of global memory transactions.

Then, Tensor Cores are introduced in ConvStencil. Due to the powerful FP64 floating point computation capabilities of Tensor Cores, the performance has improved by 76%, 68%, and 44% respectively. Next, paddings are used to reduce bank conflicts in shared memory on GPU. Paddings change the data layout across shared memory banks and remove load bank conflicts. In ConvStencil, the number of load operations in shared memory significantly exceeds the number of store operations. Although store bank conflicts still exist, we gain 1%, 14%, and 10% performance improvements in Heat-1D, Box-2D9P, and Box-3D27P, respectively. The performance improvement of Heat-1D padding is relatively inconspicuous. This is primarily attributed to the fact that the stencil2row matrices of Heat-1D contain fewer columns and load operations, thereby padding benefits outweigh overheads lightly.

However, the padding area is blank and wasted in the common padding technique. Finally, we propose dirty bits padding to utilize the area and remove conditional branches. At this stage, we witnessed a 4%, 19%, and 13% enhancement in performance metrics. At this point, we have demonstrated the effects of all optimization methods in ConvStencil.

5.3 State-of-the-art Comparison

In Figure 7, ConvStencil shows a clear performance advantage over all state-of-the-arts.

In the convolution aspect, compared with cuDNN, ConvStencil improves the performance sustainably by 2.89x on minimum and 42.62x on maximum. This result is attributed to not using Tensor Cores and not optimizing for one-channel cases. Although AMOS maps the stencil computations to the Tensor Cores, its performance is even worse than cuDNN, because it conducts a direct and unoptimized stencil-to-Tensor-Cores mapping and wastes most compute capacity of Tensor Cores.

In the stencil aspect, ConvStencil achieves an average 2.77x speedup compared to Brick. ConvStencil also achieves an overall 2.02x speedup on average compared to DRStencil. In Heat-2D and Box-2D9P, TCStencil outperforms DRStencil, but it still significantly falls behind ConvStencil. Despite using Tensor Cores, the inefficiency of TCStencil arises because its algorithm is sub-optimal, characterized by a majority of zero elements in Tensor Core computations. Besides, as shown in Table 5, the number of uncoalesced global accesses and bank conflicts per request is obviously more than that of ConvStencil, resulting in a performance decline.

5.4 Does Performance Gain Attribute to Kernel Fusion?

Although this paper does not involve temporal blocking [19, 35], we apply kernel fusion to densify computations for appropriate shapes. This subsection investigates how much performance gains from the kernel fusion technique.

Figure 8 shows the comparison between kernel fusion shapes of ConvStencil and DRStencil fusing 3 time steps (DRStencil-T3). For 2D shapes, the input scale step is 256, while for 3D shapes, the input scale step is 32. In Figure 8, ConvStencil outperforms DRStencil-T3 in the vast majority of cases. In the case of Heat-2D and Box-2D9P, ConvStencil outperforms DRStencil-T3 when the input size exceeds 768^2 and 512^2, respectively. As the performances of ConvStenci and DRStencil-T3 plateau, ConvStencil achieves speedups of 1.42x and 2.13x compared to DRStencil-T3 respectively.

Table 5. Conflicts comparison to TCStencil.

Kernels	Heat-2D		Box-2D9P	
Metrics	UGA[1]	BC/R[2]	UGA	BC/R
TCStencil	49.40%	0.91	45.35%	1.29
ConvStencil	3.42%	0.39	3.42%	0.39

[1] UGA denotes the percentage of uncoalesced global accesses.
[2] BC/R denotes the average bank conflicts per request.

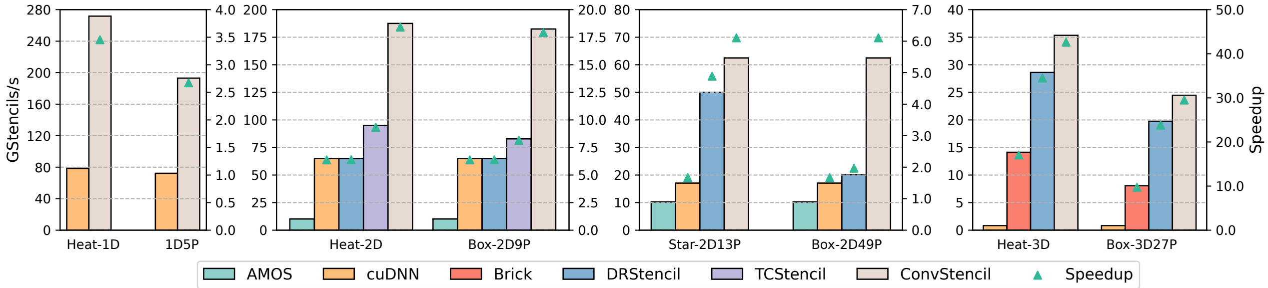

Figure 7. Performance comparison between state-of-the-arts and ConvStencil.

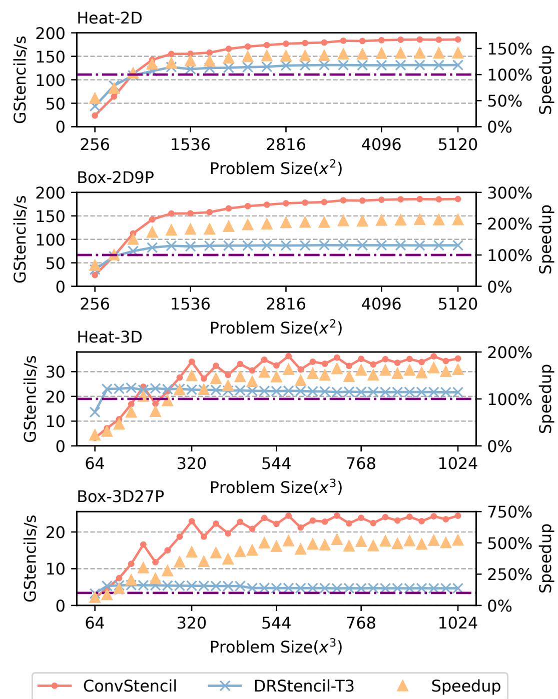

Figure 8. Performance comparison between ConvStencil and DRStencil-T3.

In the case of Heat-3D and Box-3D27P, ConvStencil outperforms DRStencil-T3 when the input size exceeds 288^3 and 128^3, respectively. After their performances stabilized, ConvStencil achieves speedups of 1.63x and 5.22x compared to DRStencil-T3 in Heat-3D and Box-3D27P. Another phenomenon is the fluctuation of ConvStencil's performance in 3D kernels. This is due to the growth step of the input scale being 32, while the spacial tiling used by ConvStencil is 64.

This paper does not involve other optimizations related to spatial and temporal blocking in DRStencil. After this comparison with DRStencil-T3, we conclude that our performance gains do not primarily originate from kernel fusion.

6 Related Work

The optimization and acceleration of stencil computation on CPU have been the subject of extensive research [23, 45]. Vectorization utilizes SIMD instructions to improve the performance of stencil computations [17, 18, 23]. Data reuse technique optimizes the order of execution instructions in order to decrease load or store operations, thus reducing the register pressure [36, 40, 49]. Tiling exploits the data locality of multiple loop nests to accelerate stencil computations, such as diamond tiling [5, 8], time skewing tiling [22, 42], rectangular tiling [39], and tessellating tiling [45].

Stencil optimizations on GPU are also widely studied [27, 33, 37]. The tiling technique is also powerful on GPUs, including spatial tiling [14, 26, 49] and temporal tiling [7, 15, 19, 29, 34, 41]. Besides, stencil optimizations on GPU include unrolling [16], prefetching [38], and streaming [37]. Brick [49–51] exploits data reuse opportunities within a fine-grained block of a stencil computation and achieves performance portability across CPU and GPU. DRStencil [43] leverages the fusion-partition optimization to accelerate the stencil computation and implements it into an effective code generation framework. The above studies focus on CUDA core, while a limited number of studies have explored Tensor Cores for stencil. To our best knowledge, TCStencil [24] is the only work that applies Tensor Cores to stencil computation. However, it is designed in FP16 precision, which limits its practicality. cuDNN [11, 31] is a library developed by NVIDIA for deep learning. It provides highly optimized implementations for primitive functions, such as convolution. AMOS [52] maps different operations from software to different hardware including Tensor Cores. It supports depth-wise convolutions that are computationally equivalent to stencil operations.

7 Conclusion

This paper introduces ConvStencil, transforming stencil computation to matrix multiplication on Tensor Cores. Inspired by GEMM-based convolution, it comprises Layout Transformation, Compute Adaptation, and Conflicts Removal. Our evaluation shows that our designs prove to be effective and ConvStencil outperforms state-of-the-arts. We believe and hope that ConvStencil promises to improve the performance of various scientific and engineering applications.

References

[1] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A Badawy. 2022. Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

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

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

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

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

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

[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] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft.

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

[12] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (March 2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[13] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 46–57. https://doi.org/10.1145/3330345.3331057

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[30] Nvidia. 2023. CUDA C++ Programming Guide. https://https://docs.nvidia.com/cuda/cuda-c-programming-guide/, Last accessed on 2023-7-24.

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

[32] Nvidia. 2023. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, Last accessed on 2023-7-24.

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

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

[35] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P. Sadayappan. 2016. Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (Barcelona, Spain) *(GPGPU '16)*. Association for Computing Machinery, New York, NY, USA, 92–102. https://doi.org/10.1145/2884045.2884047

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

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

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

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

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

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

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

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

[44] 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

[45] 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

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

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

[48] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. 2022. Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization while Ensuring QoS. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 800–813. https://doi.org/10.1109/HPCA53966.2022.00064

[49] 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

[50] 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

[51] 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

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

A Artifact Description

This artifact contains the source code of ConvStencil, a novel stencil computing system to transform stencil computation to matrix multiplication on Tensor Cores efficiently.

A.1 Availability

Our code is released on GitHub: https://github.com/microsoft/ConvStencil. Our artifact is also available on Zenodo: https://zenodo.org/doi/10.5281/zenodo.10225523.

A.2 Requirements

A.2.1 Hardware Dependencies.

- x86-64 CPU
- A single NVIDIA A100 GPU
- 20GB of memory

A.2.2 Hardware Dependencies.

- CUDA - 12.2 (Tested). Lower versions down to CUDA 11.0 are also supported, but it may affect the performance.
- GCC - 9.4.0 (Tested). You may also try to use icx or clang.
- cuDNN - above 8.0.

A.3 Getting Started

The artifact is hosted at https://github.com/microsoft/ConvStencil. Our artifact can be acquired using:

```
$ git clone https://github.com/microsoft/
    ConvStencil.git
```

Our artifact can be compiled using:

```
$ mkdir -p build
$ cd build
$ cmake ..
$ make all -j24
```

This will generate three executable files in the `build/` directory: `convstencil_1d`, `convstencil_2d`, and `convstencil_3d`.

A.4 Reproducing Results

You can run `convstencil` in the following input format:

```
$ convstencil_{x}d shape input_size
    time_interation_size options
```

- `convstencil_{x}d` can be chosen from `convstencil_1d`, `convstencil_2d`, and `convstencil_3d` for different dimensions.
- `shape` can be chosen by the different dimension:
  - `1d1r` and `1d2r` for 1D.

  - `star2d1r`, `box2d1r`, `star2d3r` and `box2d3r` for 2D.
  - `star3d1r` and `box3d1r` for 3D.
- `input_size` depends on the number of dimensions; the number of inputs required is equal to the number of dimensions.
- `time_interation_size` is the iteration time.
- `options`:
  - `--help` prints the help information.
  - `--custom` inputs the custom stencil kernel weights.

A.5 Output Format

In non-`breakdown` mode, `convstencil` will generate the following output, indicating the computation time and speed.

```
$ ./convstencil_2d box2d1r 10240 10240 10240
INFO: shape = box_2d1r, m = 10240, n = 10240,
    times = 10240
ConvStencil(2D):
Time = 17109[ms]
GStencil/s = 188.268193
```

In `breakdown` mode, `convstencil` will output the computation time and speed for different experiments.