Accelerating Complex Stencil Computations with Adaptive Fusion Strategy

Siqi Wang
Beihang University
Beijing, - None -, China
lethean1@buaa.edu.cn

Hailong Yang
Beihang University
Beijing, - None -, China
hailong.yang@buaa.edu.cn

Pengbo Wang
Beihang University
Beijing, - None -, China
wangpengbo2022@buaa.edu.cn

Shaokang Du
Beihang University
Beijing, - None -, China
shaokangdu@buaa.edu.cn

Yufan Xu
Independent Researcher
Cupertino, USA
sz.yufanxu@gmail.com

Qingxiao Sun
China University of Petroleum, Beijing
Beijing, - None -, China
qingxiao.sun@cup.edu.cn

Xiaoyan Liu
Beihang University
Beijing, - None -, China
liuxiaoyan@buaa.edu.cn

Xuezhu Wang
Beihang University
Beijing, - None -, China
wangxuezhu@buaa.edu.cn

Xuning Liang
Beihang University
Beijing, - None -, China
20373606@buaa.edu.cn

Zhongzhi Luan
Beihang University
Beijing, - None -, China
07680@buaa.edu.cn

Yi Liu
Beihang University
Beijing, China
yi.liu@buaa.edu.cn

Depei Qian
Beihang University
Beijing, - None -, China
depeiq@buaa.edu.cn

Abstract

Stencil computation is an important computational pattern widely utilized in various scientific applications, such as image processing, climate forecasting, and fluid dynamics. With the increasing demands for higher precision by scientific applications, stencil computations have become complex, containing a set of dependent stencil operators that may process multiple input grids. These stencils are referred to as *complex stencils*. For complex stencils, optimizing individual stencil operators is insufficient, and there is significant interest in developing optimization approaches across stencil operators. Existing stencil optimizations or compilers adopt the *producer-consumer fusion* of stencil operators to eliminate intermediate data access for better performance, however, neglecting the performance opportunity by exploiting inter-operator parallelism through *parallelism fusion* of stencil operators. To address the above limitation, we propose *Plasticine*, an adaptive fusion framework for complex stencil computations on GPU. We begin by introducing *parallelism fusion*, which enables the fusion of concurrent stencil operators for improved performance. Then, a novel multi-level complex stencil representation that effectively captures the characteristics of stencil programs is designed and a CNN-GNN-based model is utilized for fusion strategies selection. The experimental results show that our work, *Plasticine*, achieves consistently superior performance over state-of-the-art stencil compilers.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725768

CCS Concepts

• **Computing methodologies** → **Massively parallel algorithms**; **Neural networks**.

Keywords

Parallelism Fusion, Strategy Selection, Complex Stencil

ACM Reference Format:
Siqi Wang, Hailong Yang, Pengbo Wang, Shaokang Du, Yufan Xu, Qingxiao Sun, Xiaoyan Liu, Xuezhu Wang, Xuning Liang, Zhongzhi Luan, Yi Liu, and Depei Qian. 2025. Accelerating Complex Stencil

Computations with Adaptive Fusion Strategy. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3725768

1 Introduction

Stencil computation is identified as one of the most important computational motifs in scientific applications, which is widely applied across various domains including image processing [19], climate forecasting [10, 28, 29] and fluid dynamics [7, 13]. The data derived from these applications are regarded as structured grids. A stencil computation traverses through a grid and updates grid points with fixed neighbor points. The extent of the neighbors along each dimension is denoted as *stencil order*. Temporal iteration is involved in the update process, where the value of one point at time t is a weighted sum of itself and neighboring points at time $t-1$.

With the increasing need for higher precision in scientific applications, physical models are becoming more and more complex. This trend leads to stencils used in these applications exhibiting increasingly diverse and complex stencil orders, data accesses, and computing patterns [11, 27]. Specifically, these programs run many different stencil operators that exhibit complex dependency patterns among them. Moreover, each stencil operator is also performed at a high order with data processing from multiple input grids. We refer to the aforementioned stencil as *complex stencil*. Due to the composite dependency relationships and computational patterns, complex stencils exhibit greater parallelism, making them highly suitable for acceleration on GPUs.

In complex stencil programs, optimizing individual stencil operators is often insufficient. Instead, it's more effective to optimize dependent stencil operators or the entire stencil program as a whole. *Kernel fusion* [9, 19, 26, 34] is a popular and effective technique, which combines two or more operators into one large but equivalent kernel to potentially enhance the execution performance. The standard kernel fusion in the stencil framework combines stencil operators based on producer-consumer relations repeatedly that replace all accesses to producer results by inlining computation. Therefore, such fusion strategy is also known as *producer-consumer fusion* (Figure 1(b)). The fused kernel maintains the same thread parallelism as the original kernels. Each thread within the fused kernel sequentially integrates the instructions from the corresponding threads of the original kernels. The potential performance advantage of *producer-consumer fusion* comes from the reduced memory accesses to the intermediate results. However, it fails to exploit the parallelism among stencil operators and results in the underutilization of GPU computational resources.

This paper seeks answers to the following two questions:

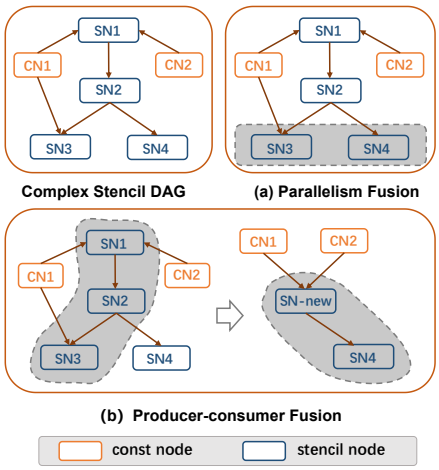

Figure 1: The complex stencil DAG (Section 3.2) and fusion strategies. The const node represents the memory container for constants and the stencil node represents the execution process of a stencil operator.

- Can a novel fusion strategy efficiently exploit the parallelism opportunities that existing methods overlooked and accelerate the execution?
- Can an approach accurately enable optimal fusion strategy selection for a specific complex stencil program?

(1) Novel fusion strategy. Inter-operator parallelism, referred to as *parallelism fusion* (Figure 1(a)), fuses independent operators and aims to execute these operators concurrently. The fused operators can better utilize GPU resources and have gained significant performance speedup in several domains such as deep learning compilation optimization [14, 17]. However, existing stencil optimization frameworks have not yet supported *parallelism fusion* for complex stencil programs [5, 9]. On one hand, unlike the deep learning domain that provides effective operator depiction of the required computation and corresponding dependencies using the compute graph, complex stencil programs currently lack a data structure that explicitly presents the inter-operator dependency relationships among them. On the other hand, unlike *producer-consumer fusion* that fuses sequential stencil computations where thread parallelism remains unchanged, *parallelism fusion* attempts to fuse concurrent stencil computations where a new mechanism needs to be designed for achieving parallel execution.

(2) Optimal fusion strategy selection. Even with *parallelism fusion* enabled, another challenge arises in determining the optimal fusion strategy between *parallelism fusion* and *producer-consumer fusion*. Due to the varied characteristics of complex stencil programs, no single fusion strategy consistently outperforms the others. For example, a complex stencil

program containing a number of concurrent stencil operators with a light workload can benefit more from *parallelism fusion*. Whereas, a complex stencil program containing stencil operators with intensive data round trips to the GPU memory can achieve better performance from *producer-consumer fusion*. In addition, as *parallelism fusion* typically results in costly kernel tuning, simply evaluating two fusion strategies will lead to prohibitive trial-and-error overhead.

However, determining the fusion strategy is challenging due to the complicated feature space of complex stencil programs. The feature space depicts intricate dependencies between operators and various attributes of the stencil operator such as neighbor count (denote computational complexity), neighbor density and position (denote data locality), etc. Accurate decision-making requires extracting essential information about computation patterns, memory access behaviors, and inter-operator dependencies, as well as learning the correlations between features and performance.

Fortunately, we argue that determining the better fusion strategy for complex stencil programs can be viewed as a classification problem (e.g., whether *producer-consumer fusion* or *parallelism fusion* should be applied). The fundamental advantage of the above transformation is that, it enables a new exploration approach by extracting the effective features of complex stencil programs and building an accurate model to determine the better fusion strategy for complex stencils.

In this paper, we propose *Plasticine*, an adaptive fusion framework that determines the better fusion strategy for complex stencil programs on GPU. *Plasticine* identifies the performance potential of fusing concurrent stencil operators in complex stencil programs and introduces *parallelism fusion* to facilitate the efficient execution of these fused operators. Additionally, *Plasticine* introduces multi-level complex stencil representations providing a concise and effective description of the complex stencil programs. Leveraging such representations, *Plasticine* builds an accurate prediction model to select the better fusion strategy of stencil operators using a combination of Convolutional Neural Network (CNN) and Graph Neural Network (GNN).

The key contributions of the paper are as follows:

- We identify the performance opportunities for fusing concurrent stencil operators within complex stencil programs and propose *parallelism fusion* to realize higher execution efficiency on GPU.
- We present novel multi-level complex stencil representations that model complex stencil programs using binary tensors and adjacency matrices. Furthermore, we design a random complex stencil generator that automatically creates a diverse set of valid stencil programs, which are utilized to train the prediction model for optimizing fusion strategies.
- We build an effective CNN-GNN-based prediction model for determining the better fusion strategy. Specifically, we use the CNN model to extract features at intra-operator level and the GNN model for capturing features at inter-operator level to generate the determined fusion strategy.
- Experimental results demonstrate that *Plasticine* delivers substantial performance improvements in throughput and hardware utilization compared to existing stencil compilers.

2 Background

2.1 Stencil and Complex Stencil

Stencil is one of the thirteen Berkeley motifs [2] and arises as a principal class of floating-point kernels in high-performance computing. The grid dimensions (1D, 2D, etc.), shapes (box, star, etc.), orders (3-point, 5-point, etc.), and dependency types (Jacobi, Gauss-Seidel, etc.) are among the most critical attributes in stencil computations. Algorithm 1 shows a 2D 9-point box stencil, where the central point is updated using itself and eight neighboring points. Recently, stencil computations have become more diverse in real-world applications [6, 30]. Each stencil program consists of multiple stencil operators with direct or indirect dependency relationships. Moreover, each stencil operator exhibits more composite computing and data access patterns at a higher order with data processing from multiple input grids. We refer to the aforementioned stencil as *complex stencil*.

Algorithm 1: 2D 9-point Box Stencil

Input: input grid I, coefficient c_1 c_9.
Output: output grid O.

1 **foreach** *point in inner region* **do**
2 $O[i][j] =$
$c_1 \times I[i-1][j-1] + c_2 \times I[i-1][j] + c_3 \times I[i-1][j+1]$
$+ c_4 \times I[i][j-1] + c_5 \times I[i][j] + c_6 \times I[i][j+1]$
$+ c_7 \times I[i+1][j-1] + c_8 \times I[i+1][j] + c_9 \times I[i+1][j+1]$

Algorithm 2 represents a snippet of the real-world complex stencil *p_grad_c* from the Finite-Volume Cubed-Sphere Dynamical Core (FV3) [6]. FV3 is the dynamical core of the CM4 and GEOS-5 global climate models and is capable of both hydrostatic and non-hydrostatic atmospheric simulations. Due to the complexity of the physical system, most of the stencils used in these models are complex stencils. As shown in Algorithm 2, each loop represents a stencil operator, and the entire algorithm consists of multiple stencil operators with dependencies. For example, the output of the first operator (i.e., *wk*) is one of the inputs of the second and third stencil operators. Moreover, each stencil operator can

involve computations of multiple grids. For example, in the second stencil operator, the computation is applied on two grids, namely *rdxc* and *wk*.

Algorithm 2: Partial p_grad_c Stencil

Input: input grid *delpc*, *rdxc*, *rdyc*...
Output: output grid *uout*, *vout*...

```
1 foreach point in inner region do
2     wk[i][j][k] = delpc[i][j][k]
3 foreach point in inner region do
4     uout[i][j][k] = rdxc[i][j][k]/
        (wk[i − 1][j][k] + wk[i][j][k])...
5 foreach point in inner region do
6     vout[i][j][k] = rdyc[i][j][k]/
        (wk[i][j − 1][k] + wk[i][j][k])...
7 ...
```

2.2 Graph Neural Networks

Graph Neural Networks (GNNs) have emerged to stand on the frontline for addressing graph-based tasks [12, 15, 33, 37]. The main objective of GNNs is to learn the node and graph representation for predicting the attributes of nodes or the entire graph. GNN takes graph-structured data as inputs, including the adjacent matrix A of the graph and feature matrix X. Each row of X represents the feature vector x_v of node v in the graph. GNN then learns the feature vectors via a two-step process including *neighbor aggragation* and *node update*, which can be formulated as:

$$\begin{aligned} a_v^{(k)} &= Aggregate^{(k)}(h_u^{(k-1)} | u \in N(v)) \\ h_v^{(k)} &= Update^{(k)}(a_v^{(k)}, h_v^{(k-1)}) \end{aligned} \tag{1}$$

where $h_v^{(k)}$ represents the embedding vector of node v at the k-th layer and $h_v^0 = x_v$, $a_v^{(k)}$ is the intermediate aggregated feature through collecting the information of neighbors, and $N(v)$ is the neighbor set of node v which is obtained from A. In the k-th layer, node v gathers its neighbors' states to an aggregation through aggregate function. The aggregated state is then combined with the previous state of node v (e.g., $h_v^{(k-1)}$) through the update function.

The aggregate function and update function may vary across different GNNs. For example, Graph Convolutional Network (GCN) [15] stands out as one of the most successful networks for graph learning, efficiently alleviating the issue of overfitting on local neighborhood structures for graphs. It uses a weighted average aggregator for the aggregate function and a single-layer perceptron for the update function. Furthermore, we choose it to predict the optimal fusion strategy due to its outstanding performance in classification tasks [36, 38] (details in Section 3.4).

3 Methodology and Implementation

3.1 Design Overview

In this section, we propose an adaptive fusion framework *Plasticine* that can determine the better fusion strategy for complex stencil programs. As shown in Figure 2, the *Plasticine* consists of four important components including parallelism fusion (Section 3.2), multi-level complex stencil representation (Section 3.3), CNN-GNN-based prediction model (Section 3.4), and random complex stencil generator (Section 3.5). Parallelism fusion is a novel optimization for accelerating complex stencil computation, which efficiently exploits the performance opportunities from inter-operator parallelism. Nevertheless, its optimization potential closely relies on features of the complex stencil program.

To better identify the optimal fusion strategy for various complex stencil programs, we propose an accurate prediction approach. The multi-level complex stencil representation transforms complex stencil programs into assigned tensors and adjacency matrices. With the transformed representations as inputs, a novel CNN-GNN-based prediction model is used to explore the optimal fusion strategy. The CNN extracts features of complex stencil programs related to computation and memory access and GNN utilizes dependency relation to further determine the fusion opportunity. Moreover, the random complex stencil generator produces a variety of complex stencil programs for training data collection.

Figure 2 illustrates the holistic pipeline of *Plasticine*. During the training process of the prediction model, the random complex stencil generator initially creates multiple complex stencil programs. The *neighbor access*, *multi-grid computation*, and *dependency* information of each generated stencil are transformed into binary sparse tensors and adjacency matrix through the proposed complex stencil representation for subsequent model training. For any given new complex stencil program, *Plasticine* converts them to internal representations as described above and accurately determines the optimal fusion strategy. If parallelism fusion is selected, *Plasticine* launches concurrent execution through a paralleled thread-block-level task and generates the high-performance program (details in Section 3.2). Otherwise, *Plasticine* adopts producer-consumer fusion leveraging the existing state-of-the-art implementation [9].

3.2 Parallelism Fusion

To fully exploit the opportunities for parallelism among stencil operators to accelerate the execution of the complex stencil program, we introduce a new stencil fusion technique

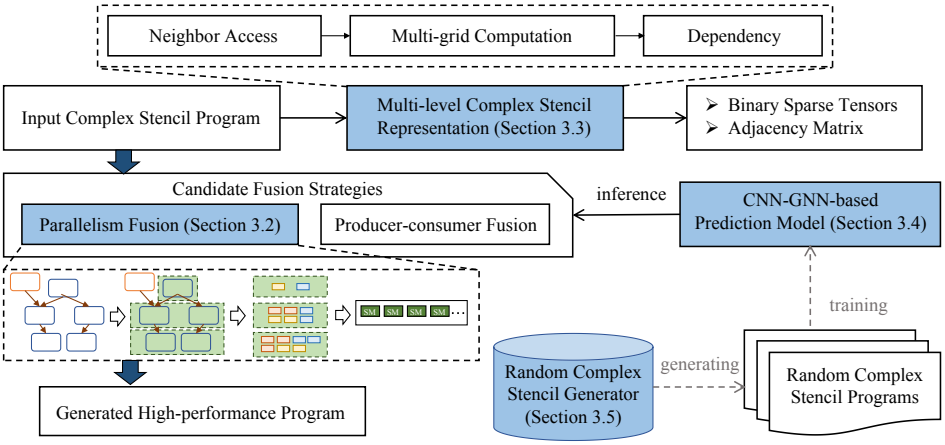

Figure 2: Design overview of *Plasticine*.

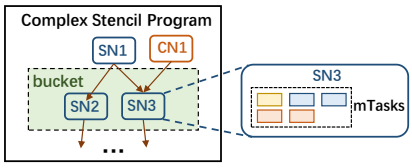

Figure 3: The illustration of the relationship among the stencil operator (i.e., *SN*), *bucket*, and *mTask*.

called *parallelism fusion*. Initially, we transform complex stencil programs into complex stencil Directed Acyclic Graphs (DAGs) to represent the dependencies among stencil operators explicitly. Subsequently, as shown in Figure 3, by utilizing the DAG, we analyze these dependencies, and group independent stencil operators (i.e., *SN* in Figure 3) to form multiple *buckets*. Finally, to enable stencil operators within a *bucket* to execute concurrently on GPU streaming multiprocessors (SMs), we introduce a thread-block-level task abstraction termed *mTask* (i.e., micro task). The detailed methodology is described as follows.

We define *mTask* as a fine-grained schedulable unit that enables the concurrent execution of multiple stencil operators. Each *mTask* is a thread-block-level task running on SMs. Through loop tiling and unrolling, we decompose the computation of a stencil operator on the entire grid into computations on individual tiles as *mTask*. Since there are no dependencies between *mTasks*, they can be executed concurrently and safely. The entire complex stencil program is thus transformed into a combination of multiple *mTasks*. By analyzing dependencies among stencil operators, we can schedule independent *mTasks* in parallel, maximizing GPU resource utilization.

To obtain the dependencies among various stencil operators within a complex stencil program, we introduce the complex stencil DAG. As depicted in Figure 4, the DAG comprises two types of node: *const node* and *stencil node*. The *const node* serves as a memory container for constants with a defined shape and initialized data, containing attributes including data type, domain size, and value. On the other hand, a *stencil node* represents the execution process of a stencil operator on the specified computational domain, with attributes including data type, computational domain size, and expression. We use Algorithm 2 as an example, *const nodes* are data objects such as *wgtfac*, *ppuv* in the computation statement at line 2; and *stencil nodes* are computation procedures of calculating *ppgk* and *ppgc* at line 2 and line 4.

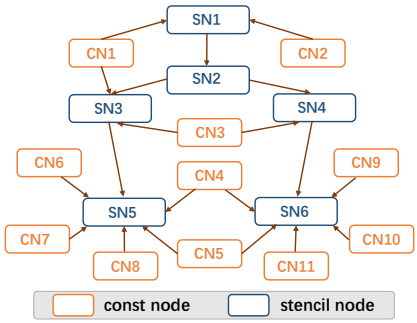

Figure 4: The illustration of complex stencil DAG.

Moreover, the directed edges connecting different nodes signify dependency relationships between operators within the complex stencil program. Each *stencil node* receives one or more inputs, which are either derived from *const nodes* or provided by the outcomes of other *stencil nodes*. For example, in Figure 4, inputs of *stencil node SN4* are the *const node CN3* and the output of *stencil node SN2*. By utilizing the DAG, we can represent the data dependencies in complex stencil programs while maintaining the original program semantics, allowing for further analysis based on this structure.

After extracting the DAG from the complex stencil program, we identify combinations of stencil operators that can be executed in parallel, with each combination referred to as a *bucket*. Specifically, as shown in Figure 5, we initially

remove the *const nodes* from the DAG to obtain the S-DAG. In S-DAG, the in-degree of a *stencil node* indicates the number of stencil operators it depends on. Nodes with a zero in-degree indicate that they can be executed safely and independently of one another, and thus, they are placed into a *bucket*. We remove the nodes placed in the *bucket* along with their outgoing edges from the S-DAG and then continue to search for the next *bucket* until the S-DAG is emptied. The process is iterative, and eventually, all stencil operators in the complex stencil program are grouped into multiple *buckets*.

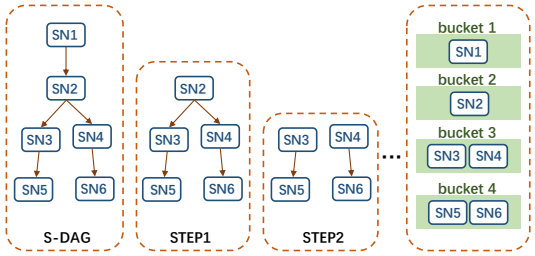

Figure 5: The illustration of *buckets* generation.

Stencil operators within each *bucket* operate independently, and likewise, the *mTasks* within each stencil operator are also independent. Therefore, all *mTasks* within the *bucket* can be executed in parallel. We consider hardware resource capacity, such as the number of threads and shared memory allocation, when scheduling and mapping *mTasks* on GPU SMs. This ensures efficient parallel execution of *mTasks* within *buckets* without violating resource constraints. To efficiently parallelize *mTasks* within the *bucket*, we leverage the backend implementation of Rammer [17]. Rammer abstracts the GPU as a virtualized parallel device, consisting of multiple virtualized execution units. Each virtualized execution unit corresponds to a physical execution unit in the GPU that performs the actual computation. We bind *mTasks* to the virtualized execution unit and achieve parallel execution among independent *mTasks* in the *bucket*.

3.3 Multi-level Complex Stencil Representation

To capture the key features of a complex stencil program and present them more concisely, we propose a new multi-level complex stencil representation.

Analysis of Complex Stencil Programs. We model the complex stencil program into three hierarchical levels of representations. First, the complex stencil program is composed of multiple stencil operators. In real-world applications, due to the complexity of physical systems, the number of stencil operators is typically more than three. These stencil operators exhibit static and acyclic dependencies between them. We encode the relationships between stencil operators within complex stencil programs as *dependency* representation.

Second, each stencil operator involves computations on multiple grids [27, 30], which define the computation and memory access pattern of the stencil operator. We describe computations within a stencil operator across multiple grid inputs as *multi-grid computation* representation.

Lastly, computations on a single grid require sweeping the grid and processing the fixed neighbors around each point to update values. Therefore, the position of neighbors affects the grid computation. We extract the computation information on a single grid as *neighbor access* representation.

In summary, a complex stencil program (CSP) can be represented as follows:

$$\begin{aligned} &CSP := (Complex\ Stencil\ DAG, Stencil\ Operators) \\ &Stencil\ Operator := \{Grid\ Access_1 \cup \ldots \cup Grid\ Access_n\} \end{aligned} \tag{2}$$

where *Complex Stencil DAG* can be encoded as *dependency* representation, the cross grid computations within *Stencil Operator* can be encoded as *multi-grid computation* representation, and the *Grid Access* can be encoded as *neighbor access* representation.

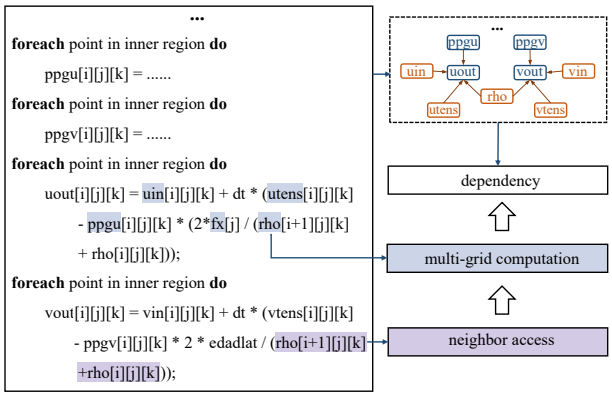

Figure 6: An example of *neighbor access* representation, *multi-grid computation* representation, and *dependency* representation.

Figure 6 shows an example of representations. For *neighbor access*, *rho* accesses the central point and its neighboring points. In *rho*'s tensor representation, these points are set to 1, whereas other points are set to 0. For *multi-grid computation*, the computation of *uout* is represented by the concatenation of tensors that represent *uin*, *utens*, *ppgu*, *fx* and *rho*. For *dependency*, the dependency between *ppgu* and *uout* is represented with the corresponding positions in the adjacency matrix setting to 1. The detailed representations are shown as follows.

Neighbor Access Representation. We show the representation of a single grid computation within the complex stencil operator here. Motivated by [4] and [32], we represent the access information of computations on each grid, regardless

of order or shape, as a binary sparse tensor. In this representation, the central point and the accessed neighboring points are assigned a value of 1, while the remaining points are assigned a value of 0. The representation of the binary sparse tensor can efficiently capture the distribution of neighbors accessed and their Euclidean distances. This information governs the load characteristics of memory-intensive stencil computations, which significantly impact the performance of stencil computations under different fusion strategies.

Muti-Grid Computation Representation. A complex stencil operator is composed of computations on multiple grids. Building upon the *neighbor access* information from individual grids as described above, we further model these representations. Initially, we collect all computations on each grid contained within the stencil operator and obtain several binary sparse tensors. Given that computations on different grids within a stencil operator are independent of each other, we concatenate these binary sparse tensors directly to obtain a high-dimensional tensor that characterizes the number of grids and the load characteristics on each grid within the entire stencil operator. This information, closely tied to the computation and memory access of the stencil operator, influences the efficiency of different fusion strategies.

Dependency Representation. The computation of a complex stencil program is composed of multiple stencil operators that have dependency relationships. We further model the dependencies between these stencil operators. As illustrated in Section 3.2, the computation of a complex stencil program can be transformed into a complex stencil DAG. Therefore, leveraging the graph structure inherent in the complex stencil program, we represent the stencil operator dependencies as an adjacency matrix, denoted as A, of size $[num(const\ node)+num(stencil\ node)]\times[num(const\ node)+num(stencil\ node)]$. If a *node i* (*const* or *stencil*) has a relationship with another *stencil node j*, the value at the corresponding points (A_{ij}) in the adjacency matrix is set to 1; otherwise, it is set to 0. We convert this adjacency matrix into a two-dimensional tensor for storage.

3.4 CNN-GNN-based Prediction Model

The deep learning models that process data modeled as graphs have been widely applied in learning node representations and relationships between nodes, which exhibits exceptional performance. Motivated by these advances, our approach models the fusion strategy selection problem as a graph classification task. Driven by the complicated feature space of complex stencil programs, we propose a CNN-GNN-based prediction model to guide the selection of strategies. Both the CNN and GNN models play a pivotal role by extracting essential information about computation patterns, memory access behaviors, and inter-operator dependencies. The CNN component is utilized to capture the influencing features at the intra-operator level, while the GNN component further extracts graph-level dependencies at the inter-operator level to predict effective fusion strategies. We discuss our design in detail as follows.

Model Design of CNN. The CNN component is utilized for exploring intricate features related to memory access and computation in stencil computation. As shown in Figure 7, we propose a lightweight network architecture. The model's input is the representation of the stencil operators and CNN produces a feature vector that represents the patterns of grid computations within the complex stencil operators.

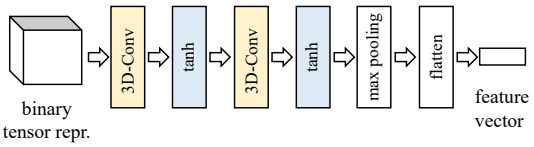

Figure 7: The model design of CNN.

Model Design of GNN. The GNN component obtains inter-operator dependencies and integrates the intra-operator features from CNN to make the prediction. We model the fusion strategy selection as a classification task on learned graph representations. There are amount of work focused on graph-based learning, and we choose the graph convolutional network (GCN) [36] as it is the most suitable network for our scenario. As shown in Figure 8, the inputs of the model are feature vectors of each stencil operator along with the adjacency matrix that represents the relationship between stencil operators. The output of the model is the probability of the complex stencil program selecting producer-consumer fusion and parallelism fusion. The model first consists of four *GCNConv* layers, each along with a *tanh* as the activation function. After passing the initial node embeddings (i.e., feature vectors) through *GCNConv* layers, we leverage resulting embeddings to learn a representation for the entire graph. Specifically, we select the *global_add_pool* operation, which involves adding the node embeddings. The consolidated feature vector produced by this pooling operation is regarded as the final output.

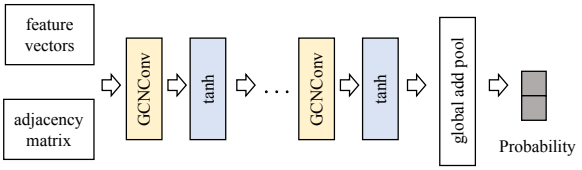

Figure 8: The model design of GNN.

CNN-GNN-based Co-design. Figure 9 represents the CNN-GNN-based prediction model and shows the workflow of processing a complex stencil program for each iteration. The model inputs consist of *multi-grid computations* of stencil

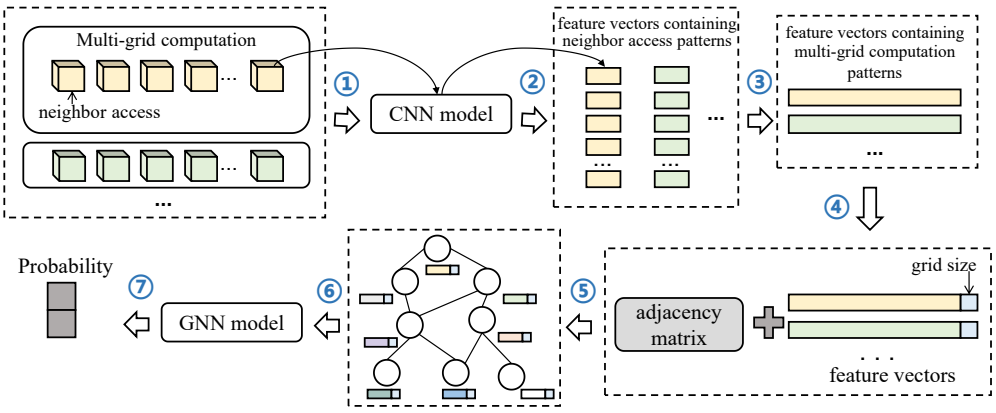

Figure 9: The design of CNN-GNN-based prediction model.

operators and the *dependencies* (i.e., adjacency matrix) representing the inter-operator relationships. The returns of the model are the probabilities of selecting producer-consumer fusion and parallelism fusion.

As shown in Figure 9, the CNN part takes *multi-grid computations* as inputs, processing each included *neighbor access* individually (①). These *neighbor accesses* are binary tensors and outputs of CNN are multiple feature vectors representing the neighbor access patterns (②). To obtain feature extraction of stencil operators, we further concatenate the above vectors associated with the same stencil operator (③) and obtain feature vectors representing multi-grid computation patterns. Moreover, we incorporate the *grid_size* of stencil operators to the end of each feature vector (④). These feature vectors serve as partial inputs required for GNN inference (⑤), and each vector represents the computation and memory access patterns of the stencil operator. Combining the adjacency matrix which represents the inter-operator dependencies, we feed both feature vectors and the adjacency matrix to the GNN model (⑥). The GNN model, after undergoing the analysis in the GCN layer, outputs the probabilities of selecting each fusion strategy (⑦).

3.5 Random Complex Stencil Generator

The random complex stencil generator aims to output various complex stencil programs for training data collection. We adhere to Section 3.3 to generate each level representation and leverage these representations as training data. Specifically, we randomly generate valid *neighbor accesses* and use them to create *multi-grid computations*. Both of them are represented as sparse tensors for each randomly sampled stencil operator and serve as partial input training data. With tensors generated in the first two steps, we can represent various types of stencil operators. Additionally, to generate complex stencil programs, we mimic the topologies of real-world complex stencil programs to create random *dependencies*. The random *dependencies* are stored in adjacency matrices which eventually are used as training data as well.

Neighbor Access Generation. As discussed above, the *neighbor accesses* are represented as binary sparse tensors where the central point and neighbor points accessed are assigned a value of 1. According to our observations, in real-world applications, the majority involve 3D complex stencils with an order (the range from neighbor points to center point) not exceeding 3. Therefore, we set the dimensions of the generated tensor to $7 \times 7 \times 7$ ($7 = 3+1+3$) where the two '3's represent the order of the left and right parts respectively, with '1' representing the central point. The *neighbor accesses* are generated by random sampling in this tensor space. We first determine the maximum order of the *neighbor access* based on a certain probability distribution. Once the order is selected, we traverse each layer within the range of the selected order starting from the central point. At each layer, we randomly determine the neighbor points of the central point. These newly selected neighbor points then serve as the central points to continue selecting points for outward expansion towards the outer layers. Ultimately, we remove duplicates to ensure the diversity of the *neighbor accesses*.

Muti-Grid Computation Generation. The *multi-grid computations* are represented as high-dimensional tensors constructed by concatenating binary tensors representing *neighbor accesses*. We further generate *multi-grid computations* based on the obtained *neighbor accesses*. First, we determine the number of *neighbor accesses* within the *multi-grid computation* through random sampling. We then concatenate compatible *neighbor accesses* randomly and form the *multi-grid computation*. Given the finite number of neighbor point accesses in the real-world stencils due to their physical characteristics, we also impose restrictions on the total number of neighbor points accessed by the *multi-grid computations*. If this limit is exceeded, regeneration is performed until the requirements are met.

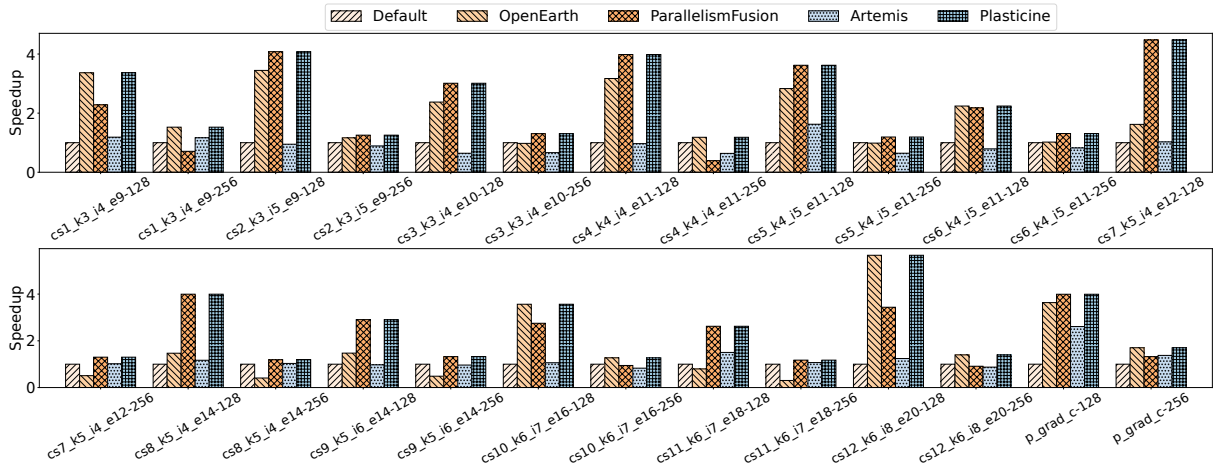

Figure 10: The overall performance comparison of different stencil compilers/frameworks on V100 GPU.

Dependency Generation. We generate *dependencies* to present the relations of stencil operators within a complex stencil program represented as adjacency matrices. To create meaningful random programs, we conduct research and analysis on the stencil topology structures in real-world stencils, extracting multiple common sub-topological chains. These sub-topological chains are randomly selected and concatenated to form *dependencies*. We validate the generated *dependencies* to ensure the validity of generating complex stencil programs. This process involved detecting cycles in the graph structure corresponding to the *dependency* and ensuring the graph is connected.

Each entry of input training data contains the above generated data and a selected fusion strategy. We evaluate fusion strategies among random complex stencil programs in the training data and select the best fusion strategy served as the label for it.

4 Evaluation

4.1 Experiment Setup

We run our evaluation on NVIDIA 32GB V100 GPU and NVIDIA 80GB A800 GPU. The V100 GPU is equipped with two Intel Xeon Gold 6330 CPUs and A800 GPU is equipped with two Intel Xeon Gold 6348 CPUs. To accurately obtain the performance of complex stencil programs, we use *cudaEvent* API to record the execution time and use the median timing from 10 runs.

We evaluate both synthetic and real-world complex stencil programs. For synthetic complex stencil programs, inspired by [16], we construct 12 complex stencil programs consistent with the observation of real-world applications [1, 5, 6, 27, 30]. Specifically, the DAG of the complex stencil program is derived from the real-world complex stencils, with the number of *stencil nodes* ranging from 3 to 6, and *const nodes* ranging from 4 to 8. For each stencil operator, we focus on 3D stencils and the number of computation grids is less than 6. Moreover, consistent with existing work on complex stencils [9], we set the grid size of 3D stencils to 128 and 256. We use the notation *cs1_k3_i4_e9-128* to represent a specific complex stencil program whose id is 1, the grid size is 128 and has 3 *stencil nodes*, 4 *const nodes*, and 9 edges. For real-world complex stencil programs, we evaluate *Plasticine* on stencil program *p_grad_c* from the FV3 application [6], which computes the three-dimensional pressure gradient.

We compare *Plasticine* with the sequential execution (*Default*) and state-of-the-art stencil compilers including the Artemis Compiler [27](*Artemis*) and the Open Earth Compiler [9](*OpenEarth*). Specifically, *Artemis* focuses on optimizing individual stencil operators such as tiling, streaming, prefetching, unrolling, and register optimization. *OpenEarth* fuses producer-consumer stencil operators for complex stencil programs. Moreover, we compare *Plasticine* with the implementation using *parallelism fusion* (*ParallelismFusion*) across all cases to highlight the performance advantages gained through the optimal strategy selection.

4.2 End-to-end Performance

Experimental Results. Figure 10 and Figure 11 shows the overall performance of *Plasticine* compared to different stencil compilers on V100 GPU and A800 GPU, respectively. The sequential execution (Default) performance is used as the baseline. *Plasticine* achieves better performance than *Default*,*OpenEarth*, and *Artemis*. Table 1 shows the max and average speedup of *Plasticine* compared to others. *Plasticine* can achieve a max speedup of 5.66× and 8.90× on V100 GPU and A800 GPU, respectively. Due to the increased computational power brought by A800 GPU, *Plasticine* has greater parallel opportunities, resulting in higher performance gains.

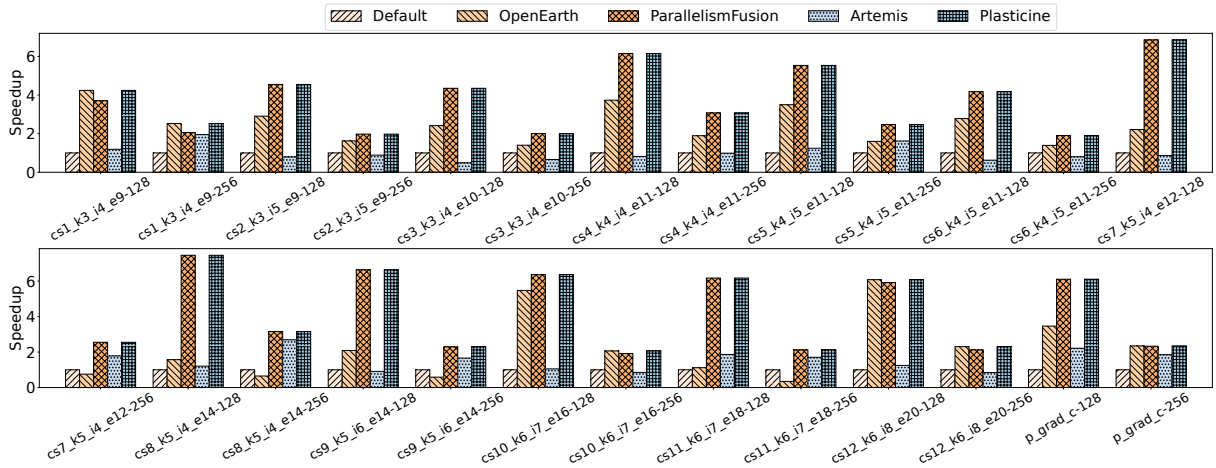

Figure 11: The overall performance comparison of different stencil compilers/frameworks on A800 GPU.

The significant performance speedup of *Plasticine* in different types of GPUs also proves its high scalability for various complex stencil programs and GPU architectures.

Table 1: Max and average speedup of *Plasticine* compared to *Default*, *OpenEarth*, *Artemis*, and *ParallelismFusion (Para-Fusion)* respectively.

		Default	OpenEarth	Artemis	Para-Fusion
V100	Max	5.66×	3.87×	4.68×	3.02×
	Avg.	2.49×	1.65×	2.39×	1.22×
A800	Max	7.46×	6.10×	8.90×	1.23×
	Avg.	4.06×	2.27×	3.86×	1.02×

Performance Analysis. *Plasticine* performs better than others for the following reasons. First, *Plasticine* proposes *parallelism fusion*, which can fully exploit the parallelism opportunities among stencil operators (Section 3.2). Moreover, *Plasticine* proposes the multi-level complex stencil representations to extract the compute and memory access features of complex stencil programs (Section 3.3). Using these representations, *Plasticine* proposes a CNN-GNN-based prediction model to determine the better fusion strategy of stencil operators (Section 3.4), and Table 2 shows the selection results. In addition, *Plasticine* also proposes a random complex stencil generator to generate various complex stencil programs for training data collection (Section 3.5). Accurate representations, thoughtful model design, and ample training data have led to the prediction model achieving a high accuracy. With the novel parallelism fusion and the ability to select appropriate fusion strategies for various stencils, *Plasticine* achieves significant performance speedup. For certain stencils like *cs1_k3_i4_e9-128*, *Plasticine* performs comparably to *OpenEarth*. This is because these stencils are better suited for producer-consumer fusion, and *Plasticine* accurately selects the better fusion strategy for them. Moreover, the speedup compared to *ParallelismFusion* demonstrates the importance of accurate fusion strategy selection.

For certain stencils such as *cs8_k5_i4_e14-256* and *c11_k6_i7_e18-256*, the performance of *OpenEarth* is slightly inferior. This is because *OpenEarth* primarily relies on the producer-consumer pattern to fuse multiple stencil operators. Although the producer-consumer fusion can reduce memory moving by reusing intermediate results, bringing performance gain, it also increases the pressure on registers and L2 cache, which may result in register spilling and increased L2 cache misses. Therefore, simply using producer-consumer fusion is unsuitable for all complex stencil programs. In some cases, it fails to deliver performance improvements and may even lead to negative optimizations.

As for *Artemis*, it focuses on optimizing individual stencil operators and conducts auto-tuning based on the register analysis of operators, while fails on complex stencil programs due to the lack of high-level information such as inter-operator dependencies. Therefore, *Artemis* always tends to utilize complex and potentially unsuitable operator optimization strategies. For example, it selects the streaming optimization for complex stencil programs with a large number of stencil operators or a high order, leading to serious register spilling issues and ultimately resulting in performance degradation.

4.3 GPU Utilization

Plasticine efficiently exploits the computational resources of GPU by leveraging parallelism fusion. We evaluate the GPU utilization improvement by *Plasticine* through comparing it with *Default*. Specifically, we use the metric *achieved_occupancy* to measure the GPU utilization, which

Table 2: Fusion strategy selection results of 13 complex stencil programs in the benchmark (*pc* refers to producer-consumer fusion and *pf* refers to parallelism fusion).

		cs1	cs2	cs3	cs4	cs5	cs6	cs7	cs8	cs9	cs10	cs11	cs12	p_grad_c
V100	**128**	*pc*	*pf*	*pf*	*pc*	*pf*	*pc*	*pf*	*pf*	*pf*	*pc*	*pf*	*pc*	*pf*
	256	*pc*	*pf*	*pf*	*pc*	*pf*	*pf*	*pf*	*pf*	*pf*	*pc*	*pf*	*pc*	*pc*
A800	**128**	*pc*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pc*	*pf*
	256	*pc*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pf*	*pc*	*pf*	*pc*	*pc*

indicates how many warps can be active at once per SMs. Higher *achieved_occupancy* means fewer idle CUDA cores and thus better GPU utilization efficiency. We evaluate the complex stencil programs that employ parallelism fusion, showcasing GPU utilization results of stencils under different grid sizes and varying numbers of stencil operators. As shown in Figure 12, compared to the *Default*, *Plasticine* can improve the GPU utilization by up to 1.48× and 1.36× with grid size 128 and 256, respectively. This indicates that a single stencil operator cannot always fully utilize GPU resources, and the improvement comes from the capability to concurrently execute multiple operators by using parallelism fusion.

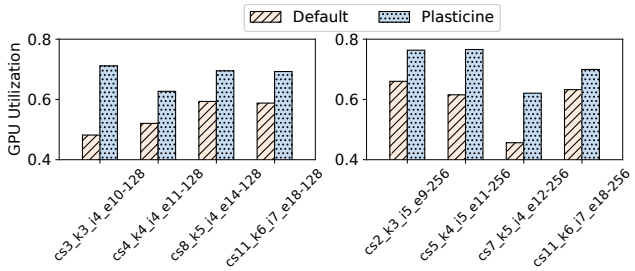

Figure 12: GPU utilization of complex stencil programs.

4.4 Prediction Accuracy

We show the prediction accuracy of our CNN-GNN-based prediction model here. Specifically, we randomly generate 3000 complex stencil programs using *Plasticine* and select 2700 as the training set, with the remaining stencils used for the test set. This is consistent with the training dataset used for the prediction model in *Plasticine*. The prediction accuracy of the prediction model on V100 and A800 GPU is 98.7% and 99.0%, respectively. This proves the practicality of representing complex stencil programs as binary tensors and adjacency matrix. Moreover, the compute and memory access characteristics of complex stencil programs can be effectively captured by convolution operations and graph operations for classification tasks.

Moreover, we have evaluated with simple models such as MLP and decision tree on a set of features such as the number of dependencies and dependence depths. The prediction accuracy drops to 66.0%/67.0% and 67.3%/66.5% on A800 and V100 GPU respectively. The above results demonstrate that simple models are inadequate for representing the characteristics of complex stencils, and thus difficult to identify optimal fusion strategies.

4.5 Network Designs

To better understand how the network design affects the prediction accuracy of the CNN-GNN-based prediction model, we first conduct a sensitivity study on the hyperparameters of the network. The variations include the hidden size of the CNN part(i.e., the final output channel) and the hidden size of the GNN part. Figure 13 shows the accuracy of the prediction model with different configurations on V100 GPU and A800 GPU. Models with different configurations are trained for 200 epochs, which is consistent with the number of epochs used in *Plasticine*. As shown in Figure 13, the proposed prediction model achieves high accuracy under different configurations and hardware architectures, and the difference in accuracy is less than 5.3%. This result indicates that our model exhibits low sensitivity to hyperparameter selection, allowing users to effortlessly choose configurations without the need for expert experience. In *Plasticine*, we select 6 as the hidden size of the CNN part and 8 as the hidden size of the GNN part for both the V100 GPU and A800 GPU.

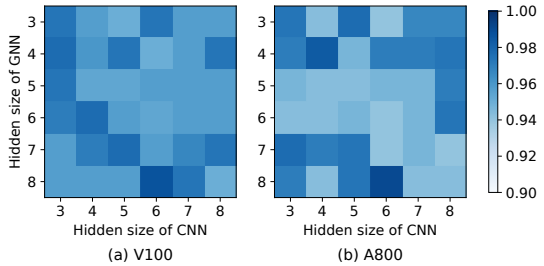

Figure 13: The accuracy of the proposed CNN-GNN-based prediction model.

To gain a better understanding of the core reasons behind the efficiency of the network, we conduct a breakdown study of the network design. We firstly disregard the GNN part of the network and solely rely on the CNN part with intra-operator features as inputs. The accuracy drops to 71.3% and 73.6% on the A800 GPU and V100 GPU, respectively.

Conversely, we disregard the CNN part of the network, and only preserve the inter-operator dependencies with GNN for model prediction. The accuracy also experienced a significant drop, reaching only 68.7% and 69.0% on the A800 GPU and V100 GPU, respectively. The experimental results show that the removal of either the CNN or GNN part severely impacts the prediction accuracy. The reason is that both intra-operator and inter-operator information are critical to represent the stencil program and affect the fusion strategy, which leads to the final choice of the network structure.

In summary, the correct model structure (e.g., the combination of CNN and GNN) plays a more important role than the structure of individual models (e.g., hidden size).

4.6 Discussion

Jointly Applying Parallelism Fusion and Producer-consumer Fusion - Currently, *Plasticine* treats the fusion optimization as a binary classification between parallelism fusion and producer-consumer fusion and thus misses the opportunities to apply hybrid fusion within a complex stencil program. We have observed the above optimization potentials exist in more complex stencils (e.g., stencils contain more than ten operators such as *horizontal diffusion* [5]). We will leave the above optimizations for future work.

5 Related Work

5.1 Performance Optimizations on Stencil

There are various works dedicated to optimizing stencil computation on GPU [18, 20, 27, 31, 39, 41]. AN5D [18] uses shared memory to cache the intermediate variables based on temporary blocking. Artemis [27] focuses on stencils with high orders and incorporates several optimizations, profiling, bottleneck analysis, and autotuning to realize high performance. These works mainly focus on optimizing single stencil operators and fail on complex stencils due to the lack of high-level information such as inter-operator dependencies.

There are also several frameworks and compilers that optimize complex stencils [5, 9]. Open Earth Compiler [9] designs stencil- and GPU-specific dialects to describe and optimize the complex stencils. However, it mainly focuses on fusing stencil operators with producer-consumer dependencies while overlooking the parallelism opportunities among stencil operators. StencilFlow [5] proposes a high-level input DSL and efficiently maps the stencil computations to multiple FPGAs. GridTools [1] is a set of libraries and utilities for applications with complex stencil computatißns at their core. However, it focuses on improving the efficiency of application development rather than optimizing code performance.

There are also other image processing frameworks such as PolyMage [19], Pluto [3], and Halide [25] support stencil computations with multiple stencil operators. However, they focus on the stencils in image processing pipelines, where individual operators exhibit dependencies in a chain-like structure, necessitating sequential processing in order. Thus, it is challenging for them to handle complex stencil programs with complicated dependencies. *Plasticine* efficiently exploits the inter-operator parallelism in complex stencil programs and achieves significant performance.

5.2 Horizontal Fusion in Deep Learning

Horizontal fusion is widely proposed to improve the performance of deep learning model workloads by exploiting the parallelism among operators [14, 17, 21, 40, 42]. Apollo [40] proposes the optimization of multi-level unified fusion, efficiently exploiting the parallelism between independent tensor operators. Astitch [42] optimizes ML computing from the aspects of dependency characteristics and memory hierarchy to maximize hardware utilization and parallelism. Rammer [17] proposes a fine-grained scheduling method to achieve efficient inter-operator and intra-operator parallelism. TASO [14] automatically generates graph substitutions, which pack independent operators into a single kernel and execute them concurrently.

However, the parallelism fusion of stencil operators is a more complex horizontal fusion of deep learning operators. The effectiveness of horizontal fusion for deep learning works can be confirmed through early profiling or domain knowledge, which is not applicable for complex stencil computation with intricate and variable characteristics.

5.3 Concurrent Kernel Execution

Previous works have proposed several techniques to support concurrent execution of kernels [8, 22–24, 35]. NVIDIA has developed Multi-Process Server(MPS) [22], which enables multiple CUDA processes to share one GPU by resource partitioning. Multi-Instance GPU(MIG) [23] has been featured since Ampere architecture GPUs, which further provides physical partitioning with multi-GPU abstraction to support concurrent execution. It partitions memory capacity, memory bandwidth, and caches in the memory system, preventing the concurrent processes from interfering with each other. However, MPS and MIG function as blind-box controllers and are unable to regulate the scheduling policies of kernels. Elastic kernel [24] identifies the lack of control over resource allocation to kernels as the major limitation of current CUDA concurrency implementation and permits fine-grained control over the resource usage of kernels. However, it statically fuses parallelizable kernels and cannot accommodate the complex stencils having various workloads.

In summary, simply utilizing existing concurrent execution approaches is insufficient to produce the optimal execution schedule and hard to accommodate complex stencil

workloads. Driven by the property of stencil workloads, we propose a fine-grained schedule abstraction(*mTask*), identifying stencil computations that can be executed in parallel, and efficiently scheduling them on the hardware. Our approach is general and can integrate with existing concurrent execution approaches.

6 Conclusion

Generating efficient fused code for complex stencil programs is challenging. In this paper, we present *Plasticine*, an adaptive fusion framework designed to determine the optimal fusion strategy for complex stencil programs. Specifically, *Plasticine* proposes *parallelism fusion* to leverage the concurrent execution opportunities among operators. Moreover, *Plasticine* provides the first evidence (to our knowledge) that a novel multi-level complex stencil representation, combined with a CNN-GNN-based prediction model, can effectively generate high-performance fusion strategies for complex stencil programs. The experimental results show that *Plasticine* can outperform the state-of-the-art stencil compilers with higher throughput and better hardware utilization.

Acknowledgments

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB3001501), National Natural Science Foundation of China (No. 62322201, U23B2020 and 62402525), and the Fundamental Research Funds for the Central Universities (YWF-23-L-1121, JKF-20240198 and JK2024-58). Hailong Yang is the corresponding author.

References

[1] Anton Afanasyev, Mauro Bianco, Lukas Mosimann, Carlos Osuna, Felix Thaler, Hannes Vogt, Oliver Fuhrer, Joost VandeVondele, and Thomas C Schulthess. 2021. GridTools: A framework for portable weather and climate applications. *SoftwareX* 15 (2021), 100707.

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

[3] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[4] Biagio Cosenza, Juan J Durillo, Stefano Ermon, and Ben Juurlink. 2017. Autotuning stencil computations with structural ordinal regression learning. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 287–296.

[5] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. StencilFlow: Mapping large stencil programs to distributed spatial computing systems. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 315–326.

[6] GFDL. 2020. FV3: Finite-VolumeCubed-SphereDynamicalCore, https://www.gfdl.noaa.gov/fv3/.

[7] GP Glinski, C Bailey, and KA Pericleous. 2001. A non-Newtonian computational fluid dynamics study of the stencil printing process. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 215, 4 (2001), 437–446.

[8] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. 2012. {Fine-Grained} Resource Sharing for Concurrent {GPGPU} Kernels. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*.

[9] 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 Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–23.

[10] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. 2015. STELLA: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–12.

[11] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 100–112.

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[13] HT Huynh, Zhi J Wang, and Peter 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.

[14] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[15] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[16] Jakub Lichman. 2020. *Learning From Generated Stencil Programs*. Master's thesis. ETH Zurich.

[17] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[18] 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.

[19] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 429–443.

[20] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[21] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN*

International Conference on Programming Language Design and Implementation. 883–898.

[22] NVIDIA. 2012. NVIDIA: Sharing a gpu between mpi processes: multiple-process service, https://docs.nvidia.com/deploy/mps/index.html.

[23] NVIDIA. 2022. NVIDIA: Nvidia multi-instance gpu user guide, https://docs.nvidia.com/datacenter/tesla/mig-user-guide.

[24] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 407–418.

[25] 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. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[26] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2018. Domain-specific optimization and generation of high-performance GPU code for stencil computations. *Proc. IEEE* 106, 11 (2018), 1902–1920.

[27] 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)*. IEEE, 641–652.

[28] U Schättler, G Doms, and C Schraff. 2008. A description of the nonhydrostatic regional COSMO-model part VII: user's guide. *Deutscher Wetterdienst Rep. COSMO-Model* 4 (2008), 142.

[29] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2020. NERO: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 9–17.

[30] STC. 2020. Consortium for Small-scale Modeling, http://www.cosmo-model.org/.

[31] Qingxiao Sun, Yi Liu, Hailong Yang, Zhonghui Jiang, Xiaoyan Liu, Ming Dun, Zhongzhi Luan, and Depei Qian. 2021. cstuner: Scalable auto-tuning framework for complex stencil computation on gpus. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 192–203.

[32] Qingxiao Sun, Yi Liu, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian. 2022. Stencilmart: Predicting optimization selection for stencil computations across gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 875–885.

[33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[34] Mohamed Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202.

[35] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 358–369.

[36] Yu Xie, Chuanyu Yao, Maoguo Gong, Cheng Chen, and A Kai Qin. 2020. Graph convolutional networks with multi-level coarsening for graph classification. *Knowledge-Based Systems* 194 (2020), 105578.

[37] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[38] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7370–7377.

[39] Liang Yuan, Hang Cao, Yunquan Zhang, Kun Li, Pengqi Lu, and Yue Yue. 2021. Temporal vectorization for stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[40] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems* 4 (2022), 1–19.

[41] 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*. 1–44.

[42] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 359–373.