Adaptive Auto-tuning Framework for Global Exploration of Stencil Optimization on GPUs

Qingxiao Sun, Yi Liu, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian

Abstract—Stencil computations are widely used in high performance computing (HPC) applications. Many HPC platforms utilize the high computation capability of GPUs to accelerate stencil computations. In recent years, stencils have become more diverse in terms of stencil order, memory accesses and computation patterns. To adapt diverse stencils to GPUs, a variety of optimization techniques have been proposed. Due to the diversity of stencil patterns and GPU architectures, no single optimization technique fits all stencils. Therefore, stencil auto-tuning mechanisms have been proposed to conduct parameter search for a given combination of optimization techniques. However, parameter search for an inappropriate optimization combination (OC) misses the globally optimal solution. To address the above problems, we propose *GSTuner*, an adaptive auto-tuning framework that efficiently determines the optimal parameter setting of the global optimization space for stencils on GPUs. Specifically, *GSTuner* represents stencil patterns as neighboring features and unifies feature vectors of OCs through data pre-processing. In addition, *GSTuner* samples parameter settings from superior OCs via the quota-based reward policy and regression mechanisms. After that, *GSTuner* employs the genetic algorithm that considers sub-population similarity to reduce the cost of evolutionary search. The experiment results show that *GSTuner* can identify better performing settings with higher auto-tuning speed compared to the state-of-the-art works.

Index Terms—Stencil Computation, GPU, Auto-tuning, Performance Prediction, Deep Learning, Genetic Algorithm.

1 Introduction

Stencil computation is one of the most adopted computation patterns in scientific applications. Stencil computations appear in many domains such as cellular automata [1], physical simulation [2] and image processing [3]. A stencil computation sweeps a computation grid and accesses fixed neighbors around each point to update its value, where the extent of the neighbors along each dimension is referred to as the *stencil order* [4]. For instance, box-shape stencils are used to perform smoothing and other neighbor-pixel-based computations in image processing [5], [6].

In recent years, stencil computations have become more diverse in terms of stencil order, data accesses and computing patterns [7], [8]. The diverse stencils tend to have abundant parallelism, which makes GPU a good candidate for performance acceleration. However, due to the complexity of GPU architecture, the programmers must ensure memory coalescing, reduce thread divergence and trade off between parallelism and resource utilization when optimizing stencils on GPU. Many optimization techniques based on streaming and tiling [9], [10] have been proposed to adapt to the high computation capability and limited memory bandwidth of GPU architecture. However, no single optimization technique fits all stencils due to the diversity of stencil patterns [11].

- *Qingxiao Sun, Yi Liu, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, Depei Qian are with Sino-German Joint Software Institute, the School of Computer Science and Engineering, Beihang University, Beijing, China, 100191.*
  Qingxiao Sun is also with Super Scientific Software Laboratory, College of Information Science and Engineering, China University of Petroleum-Beijing, Beijing, China, 102249.
  Email: {qingxiaosun,yi.liu,hailong.yang,jiangzhh,07680,depeiq}@buaa.edu.cn.

Stencil domain-specific languages (DSLs) explore the automatic code generation with the integration of optimization techniques [12], [13], [14], [15]. Although the DSLs are effective in improving stencil performance, it is difficult to evaluate the performance impact of individual optimization techniques within a particular optimization combination (OC). In addition, stencil auto-tuning frameworks [16], [17] have been proposed to determine the optimal parameter settings for specific OCs. However, whether an OC can generate high-performant code depends on the target stencil and GPU architecture. Conducting a time-consuming parameter search for sub-optimal OCs will significantly deteriorate the effectiveness of auto-tuning mechanisms. Therefore, it is necessary to explore the global optimization space to avoid missing the optimal parameter setting [18].

Performance prediction is often used to study the impact of optimizations on stencil computation [19], [20]. Since no actual execution is required, performance prediction can efficiently reduce the search cost involved in stencil auto-tuning. However, it is challenging to extract the effective features representing stencil neighboring patterns. In addition, the pre-processing of varying-length feature vectors should be conducted to handle inconsistent optimizations among different OCs. Since the performance similarity of superior settings leads to inevitable prediction errors [21], it is almost infeasible to accurately determine the optimal setting through performance prediction. Instead, performance prediction is generally utilized to guide filtering out inappropriate settings to avoid massive measurements on actual hardware [22].

To address the above challenges, we propose an adaptive auto-tuning framework *GSTuner*, which efficiently determines the optimal parameter setting of the global optimization space for stencil computation on GPUs. *GSTuner*

first generates stencil programs that satisfy stencil patterns with symmetric neighbor accesses. After that, *GSTuner* collects the stencil datasets and trains regression models for the global optimization space through data pre-processing. Then, *GSTuner* iteratively samples parameter settings from OCs, where the sampling ratios are adjusted according to the prediction results. Finally, *GSTuner* performs the search process for the sampled settings via a customized genetic algorithm with termination conditions. We evaluate *GSTuner* on various typical stencils to prove its effectiveness in performance auto-tuning on GPUs.

This paper is an extension of our previous works [11], [21]. Compared to [11], [21], we further implement a holistic pipeline for global exploration of stencil optimization. In addition, we overcome the limitations of [11], [21] by making significant design improvements in each critical component. *GSTuner* is open-sourced at https://github.com/sunqingxiao/GSTuner . Specifically, this paper makes the following contributions:

- We comprehensively analyze the impact of optimization selection on the stencil performance due to diverse access patterns. We also discuss the distribution of high-performance parameter settings among OCs and their adaptability for grid sizes.
- We propose a stencil transformation mechanism that extracts stencil features and unifies feature vectors of OCs through data pre-processing. We also offer a stencil generator that outputs a variety of stencils that satisfy symmetric neighbor accesses.
- We design a search space narrowing mechanism that samples parameter settings from OCs. The sampling process is guided by deep learning-based regression models, where the sampling ratios are determined according to a quota-based reward policy.
- We implement an evolutionary search mechanism with a customized genetic algorithm. The genetic algorithm reduces the search cost by adopting the subpopulation similarity as the termination condition.
- We develop an adaptive stencil auto-tuning framework *GSTuner* that efficiently determines the optimal parameter setting of the global optimization space on GPUs. The experiment results show that *GSTuner* can identify better performing settings in a shorter time compared to the state-of-the-art works.

The rest of this paper is organized as follows: Section 2 and Section 3 present the background and motivation. Section 4 presents the details of *GSTuner* design. Section 5 presents the evaluation results of *GSTuner*. Section 6 discusses the related work, and Section 7 concludes this paper.

2 BACKGROUND

2.1 GPU Architecture and Execution Model

The GPU consists of dozens to hundreds of Streaming Multiprocessors (SMs) depending on the GPU generation. Each SM contains hundreds of computing cores and other resources such as registers, shared memory and L1 cache. The code executed on the GPU is called *kernel*. When a kernel is launched on the CPU host, thousands of threads are created on GPU and every 32 threads are grouped into a *warp*. Furthermore, multiple warps are grouped into a *thread block* (TB), and the size of a TB is determined by kernel configuration. The TB scheduler dispatches TBs to SMs according to the Round-Robin policy, which maximizes GPU occupancy under resource and hardware constraints.

Due to the limited computing resources in SMs [23], GPU tasks have to be fine-tuned to achieve a tradeoff between system utilization and performance speedup. For instance, some optimization strategies (e.g., loop unrolling) increase register-level data reuse to improve performance [24]. However, the resulting code is highly constrained by register pressure and even causes register spilling. In addition, parameter settings need to be carefully determined to achieve better performance. For instance, an appropriate TB size maximizes thread-level parallelism (TLB) within hardware constraints. However, high TLB may cause cache thrashing, especially for memory-intensive tasks such as stencils [11]. Therefore, it is not enough to conduct performance tuning of GPU tasks through analytical modeling alone [13].

2.2 Optimizations for Stencil Computation

Widespread attention has been draw to accelerate stencil computation on GPU due to its high computation capability [5], [8], [10], [13]. We briefly discuss the optimizations of stencil computation on GPUs (Table 1).

TABLE 1: The stencil optimizations on GPUs.

No.	Optimzation	Abbr.	Constraint
1	Streaming	ST	–
2	Block Merging	BM	Not valid when CM enabled.
3	Cyclic Merging	CM	Not valid when BM enabled.
4	Retiming	RT	Only valid when ST enabled.
5	Prefetching	PR	Only valid when ST enabled.
6	Temporal Blocking	TB	–

2.2.1 Streaming

Streaming is a commonly used optimization that improves data reuse and reduces computation redundancy along the streaming dimension. For 3-D input grids, an effective implementation of streaming is 2.5-D spatial blocking [13]. Specifically, the computation of 2-D tiles is streamed over one dimension, and the data of each tile is reused for updating the next tiles. However, given large problem size, streaming increases computation granularity thus limiting parallelism. To achieve better performance, concurrent streaming [8] divides the streaming dimension into tiles, where the TBs traverse the streaming dimension in parallel at the granularity of tiles. Meanwhile, loop unrolling has been applied to increase register-level data reuse.

2.2.2 Block/Cyclic Merging

Naively, each GPU thread works on a single output point. Merging the computations of several output points reduces the overhead of kernel launching and eliminates duplicated memory accesses. Two strategies have been proposed for merging computations such as block merging and cyclic merging. For block merging, a number of adjacent output points are merged. Whereas for cyclic merging, every two points are merged with a fixed distance. However, both

strategies may increase the register pressure and reduce the number of threads that reside on each SM, thus hurting parallelism. Furthermore, block merging in the innermost dimension of the global grid can disrupt memory coalescing [17]. In general, the choice of merging strategy and the number of points to merge can significantly impact the stencil performance.

2.2.3 Prefetching

In streaming optimization, after updating the output grid, the data located in the shared memory is shifted to continue the computation for the next iteration. Due to the concurrent execution of massive threads on GPU, a synchronization barrier has to be performed between adjacent iterations to ensure the correctness of the results. The synchronization can cause serialization between kernels and thereby deteriorate performance. Prefetching [8] can hide the delay of synchronization by overlapping the computation and data loading. Specifically, the data used for the next iteration is loaded into registers simultaneously with the computation of the current iteration. However, prefetching may exhaust the registers that are quite limited on GPU.

2.2.4 Retiming

Retiming [25] improves data reuse by decomposing a stencil computation into a set of sub-computations along with accumulations. Retiming can balance the resource usage between memory and registers by homogenizing stencil accesses [8]. In general, high-order stencils can benefit from retiming optimizations due to the effective reuse of registers. However, retiming may not improve the performance of stencils with low register pressure.

2.2.5 Temporal Blocking

Even though stencil computation has data dependency across time steps, the dependency range of one point is limited by the stencil pattern and the number of time steps elapsed since the point's last update [13]. Temporal blocking exploits hidden temporal locality by fusing time steps and avoiding global memory accesses. The dependency along the time dimension is resolved by redundantly loading from adjacent blocks. However, temporal blocking may incur performance degradation for register-constrained stencils.

The above optimizations can be combined under certain constraints (Table 1) to improve performance further. The optimizations should be carefully determined to adapt the target stencil to hardware architecture. Further, inappropriate parameter settings under specific OCs inevitably lead to performance degradation. This motivates our work for the global exploration of stencil optimization with auto-tuning mechanisms that take both optimization strategies and parameter settings into account.

2.3 Limitations of Stencil Auto-tuning Mechanisms

Due to the diversity of stencil patterns, any optimization has to be fine-tuned to maximize its performance. Stencil Domain-Specific Languages (DSLs) expose performance-related parameters to auto-tuning mechanisms integrated into their frameworks [6], [8], [12]. For instance, *Halide* [12] applies stochastic search to find good pipeline schedules automatically. *Artemis* [8] tunes the computation for high-impact optimizations first and then selects a few high-performance candidates. *GoPipe* [6] finds the best task granularity for each stage of a pipelined box stencil (e.g., image convolution). Since the auto-tuning mechanisms are customized for particular stencil DSLs, they have poor scalability to evaluate more optimizations during parameter tuning.

To overcome the limitation, several works have considered speeding up the auto-tuning performance of stencil computation [16], [17], [21]. *OpenTuner* [16] implements a collection of search techniques (e.g., differential evolution and hill climber) to find the optimal solution. *Garvey* [17] groups optimization parameters based on experience and exhaustively searches for the parameter settings of each group with random sampling enabled. *csTuner* [21] leverages statistics and machine learning methods to generate parameter groups and sampled settings. Then, *csTuner* redesigns the genetic algorithm with approximation to reduce the search time. The stencil auto-tuning mechanisms usually perform parameter search for pre-specified optimizations or their combinations. This limits stencil computation to local optima, whereas the tuning results under different OCs may exhibit large performance discrepancies.

In addition to parameter auto-tuning, performance prediction is utilized to study the impact of optimizations on stencil computation [11], [19], [20]. Martínez *et al.* [19] fed the kernel configurations and hardware counters to the support vector machine (SVM) to predict the GFLOPS and execution time of stencils. Cosenza *et al.* [20] utilized ordinal regression to predict the performance ranking of stencil code variants and the quality of the obtained ranking is evaluated by Kendall coefficients. *StencilMART* [11] represented the stencil patterns as binary tensors via tensor assignment, and predicted the best OC with the convolutional neural network (CNN). However, the above works lack effective combination with parameter auto-tuning and exhibit large search cost for exploiting stencil optimization space. In addition, actual measurements are required to refine the performance models to tolerate possible prediction errors [22].

3 Motivation

We make four main observations by comparing the performance of optimization combinations (OCs), where any combination of optimizations under the constraints (Table 1) is taken into consideration. The representative stencils we select cover a variety of shapes (star, box and cross), orders (1-4) and dimensions (2-D and 3-D). The input grids of 2-D and 3-D stencils are $8,192^2$ and 512^3, respectively. We randomly sample more than 10,000 parameter settings for each stencil to conduct the motivation experiments. For each OC, the parameter setting with the shortest execution time is selected for performance comparison among OCs.

3.1 Performance Gap among OCs

Figure 1 shows the performance discrepancy between the worst OC and the best OC for each stencil on V100 GPU. Note that there are some cases where OC crashes under certain stencils, which are not reflected in the figure. For example, temporal blocking fails to be applied for 3-D

order-4 stencils without streaming enabled. This is because streaming can effectively avoid intra-SM resource spilling that may be caused by temporal blocking. As seen, the performance gap among OCs is significant, where the best OC achieves an average speedup of 9.95× over the worst OC. In addition, for stencils of the same shape, a higher dimension or order usually means a larger performance gap. However, manually determining the high-performance OCs requires considerable engineering efforts. Therefore, the performance of different OCs should be predicted in advance to avoid parameter auto-tuning under poor OCs.

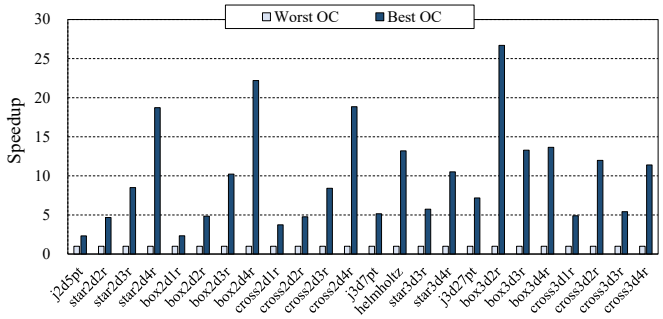


Fig. 1: The speedup of the best OC of each stencil over its worst OC on V100 GPU.

3.2 Similarity of High-performance OCs

Figure 2 shows the performance comparison of the second-best OC and the best OC for each stencil on V100 GPU. As seen, the optimal parameter settings under high-performance OCs are similar, where the second-best OC achieves an average speedup of 0.99× over the best OC. Even for the stencil with the largest performance difference (i.e., *cross3d4r*), the speedup of 0.94× is achieved. The reason is that high-performance OCs usually contain a subset of optimizations that have a large impact on performance. In such cases, roughly predicting the best OC by classification algorithms may lead to low prediction accuracy due to OC similarity [11]. It is more reasonable to predict the performance of parameter settings via regression algorithms, and then sample the settings unevenly from the OCs for actual measurements. This way, we can better balance the tradeoff between search cost and achieved performance.

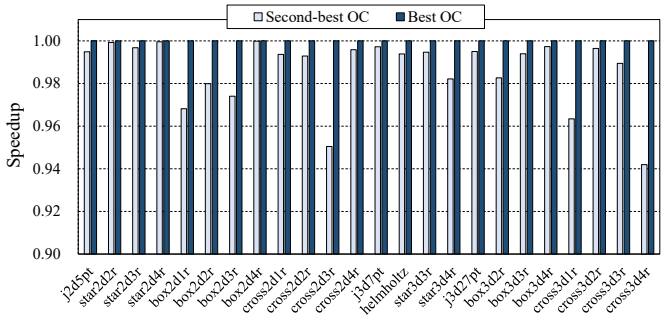


Fig. 2: The performance of the second-best OC of each stencil over its best OC on V100 GPU.

3.3 Adaptability of Optimization Settings for Grid Sizes

We randomly select six stencils to explore whether high-performance parameter settings are adaptable for grid sizes. For each stencil, we randomly sample more than 1,000 parameter settings and collect top-100 parameter settings with small grid sizes ($2,048^2$ for 2-D, and 128^3 for 3-D). The top-100 parameter settings are then applied to other grid sizes ($4,096^2$ and $8,192^2$ for 2-D, 256^3 and 512^3 for 3-D). Figure 3 shows the speedup distribution of parameter settings in ascending order of performance over the optimum for each grid size. It can be observed that the speedup curves of different grid sizes for each stencil are similar. For 2-D stencils, the best parameter setting for $4,096^2$ and $8,192^2$ sizes achieve the speedup of 1× and 1× over the optimum, respectively. Whereas for 3-D stencils, the best parameter setting for 256^3 and 512^3 sizes achieve the speedup of 1× and 0.94× over the optimum. The above results indicate that the parameter settings adapted to the target stencil and underlying hardware are relatively stable within a certain range of grid sizes. Since only the relative performance of parameter settings is required for search space sampling, the trained model of one grid size can be applied to other sizes without re-collecting the training data.

3.4 Pairwise Correlation of OCs

We define that high correlation corresponds to the small difference in performance achieved by pairwise OCs under the same stencil. Further, this indicates that the effect of pairwise OCs on stencil computation is similar. We use the Pearson correlation coefficient (PCC) [26] to quantify the correlation between pairwise OCs. The closer to 1 the absolute value of PCC is, the stronger the correlation of the OC pair is. Figure 4 shows the value distribution of top-100 PCCs achieved by pairwise OCs on GPUs of various architectures including 2080Ti, P100, V100 and A100. As seen, the value distribution of top-100 PCCs is close, and the intersection of pairwise OCs under all architectures accounts for 28% of the total. This indicates that the influence of certain OCs on stencil computation is general among architectures. Therefore, we can empirically group the OC pairs in the intersection to reduce the optimization space range. The OC with the best prediction performance is selected from each group for the sampling process.

4 GSTuner Methodology

4.1 Design Overview

In this section, we propose an adaptive stencil auto-tuning framework *GSTuner* that determines the optimal parameter setting through a holistic pipeline with the global exploration of optimization space. As shown in Figure 5, *GSTuner* consists of four important components including random stencil generator (Section 4.2), regression mechanisms (Section 4.4), search space sampling (Section 4.5) and evolutionary search (Section 4.6). Up arrows (↑) denote new components proposed by *GSTuner*, whereas up-right arrows (↗) denote components extended from previous works [11], [21]. The random stencil generator outputs a variety of stencil programs for training data collection. The regression mechanisms predict the performance under each

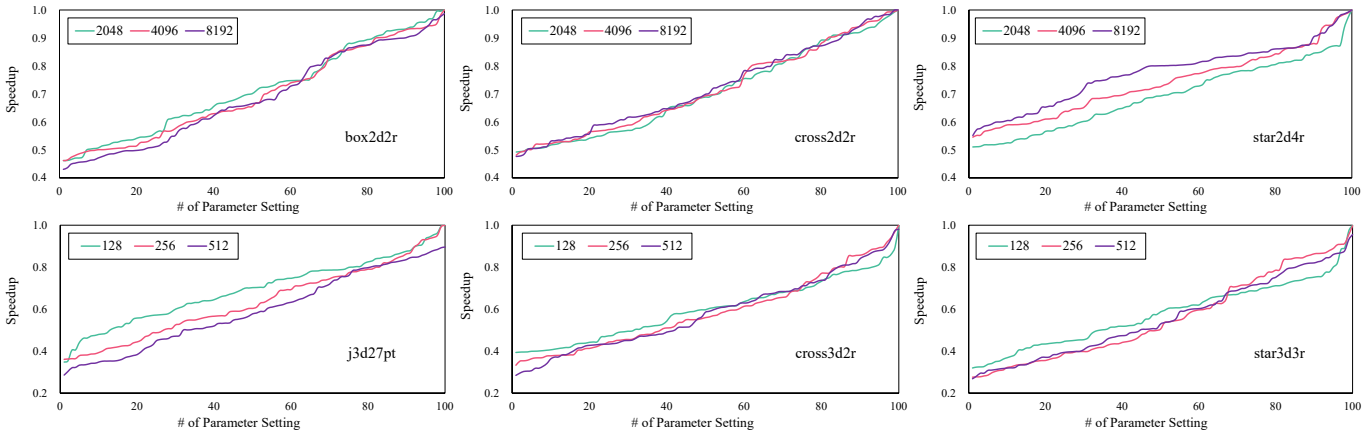


Fig. 3: The speedup distribution of the top-100 parameter settings for $2,048^2$ and 128^3 grid sizes over the optimum for other 2-D ($4,096^2$ and $8,192^2$) and 3-D (256^3 and 512^3) grid sizes.

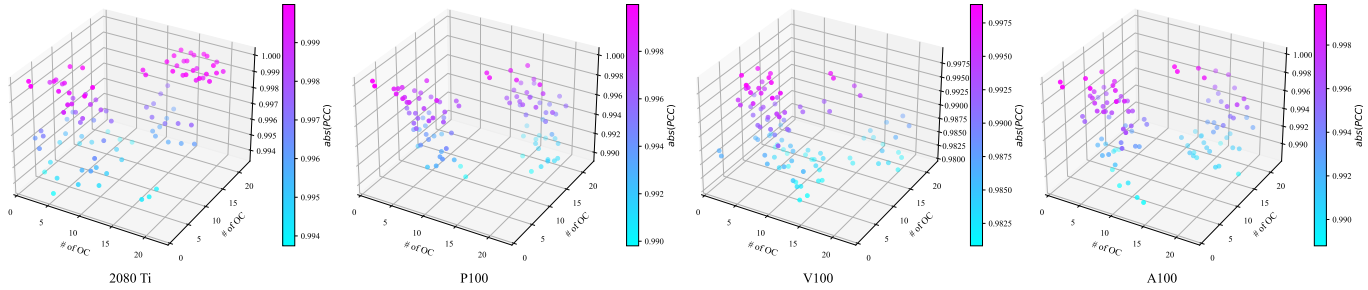


Fig. 4: The value distribution of top-100 PCCs achieved by pairwise OCs on GPUs.

unique parameter setting after data pre-processing. The search space sampling designs a quota-based reward policy to select parameter settings in high-performance OCs. The evolutionary search finds the optimal parameter setting via the genetic algorithm with termination conditions.

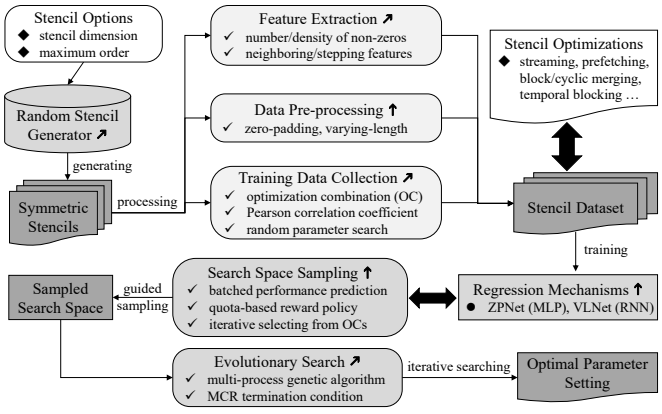


Fig. 5: The design overview of *GSTuner*.

Figure 5 illustrates the holistic pipeline of *GSTuner*. The access pattern of each generated stencil is transformed into neighboring features through feature extraction. The parameter settings in each OC are randomly searched to profile the performance of each stencil input. During this period, *GSTuner* exploits data pre-processing to address the inconsistency of vector lengths across different OCs. The profiled dataset is used to train the regression model, which guides the process of search space sampling without actual execution. *GSTuner* iteratively selects parameter settings from OCs, where the sampling ratio of OCs in each iteration is adjusted according to the reward policy. The genetic algorithm performs auto-tuning with termination conditions regarding the sub-population similarity. This eliminates the need to manually set the number of iterations based on experience, thus improving the efficiency of auto-tuning.

Since *GSTuner* inherits the strengths of our previous works [11], [21], we expect *GSTuner* to be well-suited for both local and global stencil optimization exploration on any GPU architecture. The *GSTuner* pipeline can be extended to incorporate more optimization parameters capturing future stencil optimizations. At this time, only the regression model needs to be retrained and the subsequent components can be reused. In addition to stencil computation, *GSTuner* can also support auto-tuning of more general GPU algorithms due to the versatility of its components.

4.2 Random Stencil Generation

Inspired by [20], we represent the access pattern of a stencil with any dimension or shape as a sparse tensor. Figure 6 shows an example of transforming a 2-D stencil with a maximum order of 4 into a sparse tensor with a size of 9×9. The higher-dimensional stencils can be analogized in the same way. We consider sampling access points in the tensor space to generate random stencils for model training. In this regard, the most straightforward solution is to sample within the index range of a fixed-sized tensor randomly. However, this solution does not conform to the computation

characteristic of stencils that processes the neighbors of each point to update its value. To address this issue, we design a stencil generator based on the adjacency property in *StencilMART* [11]. However, it still lacks consideration of symmetric patterns for common stencils [8], [13].

Algorithm 1 Stencil generator with symmetric patterns.

```
1: Input: stencil order (N), dimension (Dim), tensor (Tensor)
2: Output: The list of neighbor points accessed (npList)
3: LR = Tensor / pow(2, Dim) // local regions for sampling
4: for order in range [1, N] do
5:     if order == 1 then
6:         // randomly sample neighbors of central point from LR_1
7:         selected_order = central.neighbors.random(LR_1)
8:     else
9:         // randomly sample neighbors of low-order selected points
10:        selected_order = selected_{order-1}.neighbors.random(LR_1))
11:        // delete the sampled low-order neighbor points
12:        selected_order.delete(neighbor_{order-1})
13:        if order > 2 then
14:            selected_order.delete(neighbor_{order-2})
15:        end if
16:    end if
17: end for
18: for rID in range [2, pow(2, Dim)] do
19:    // Symmetrically map sampled points to other local regions
20:    selected_order.symmetry(LR_rID)
21: end for
22: // store non-redundant neighbor points to the list
23: npList.append(set(selected_order))
```

Algorithm 1 illustrates the process of random stencil generator that meets the symmetric patterns, where the input includes the stencil order, stencil dimension and fixed-sized tensor. The output is a list of neighbor points accessed by a stencil. Specifically, we partition the tensor space into local regions with overlapping boundaries and pick one of them for sampling (Line 3). After that, we iteratively sample access points from low-order neighbors to high-order neighbors within the local region. During each iteration, we randomly sample the higher-order neighbors of the selected points in the previous iteration (Lines 4-17). Then, we symmetrically map the sampled points to other local regions of the tensor space (Lines 18-21). Finally, we remove redundant neighbor points generated during sampling and mapping from the list (Lines 22-23). The random stencils generated in this way cover the popular stencil shapes (Figure 6) and conform to the symmetric neighbor access patterns.

4.3 Stencil Representation

As shown in Figure 6, we convert the offset of the accessed neighbor points from the central point into the location of the non-zero elements of a tensor. The representation of a sparse tensor captures the distribution of neighbors accessed. Unlike *StencilMART* which generates binary tensors, we assign the location of each non-zero element the Manhattan distance from the central element. This type of information largely dominates the latency of memory operations, which in turn significantly impacts the performance of stencil computation under certain optimizations. *StencilMART* feeds the assigned tensor of a fixed size to the convolutional neural network (CNN) to predict the best OC of a stencil. However, such aggressive pruning is likely to miss the global optimum that is surrounded by poor parameter settings [18]. Instead, *GSTuner* predicts the performance of certain parameter settings with regression algorithms that guide the search space sampling process.

Regression algorithms are usually combined with feature engineering to achieve better fitting results [27]. As shown in Table 2, we extract the candidate feature set according to the computation patterns of stencils. Different from [28], the candidate features extracted by *GSTuner* focus on the distance between neighbor points and the central point instead of the sparsity distribution in the entire tensor space. For example, the feature set includes the number and ratio of neighbor points of each order. The step-m indicates that the Manhattan distance between neighbor points and the central point is m. Note that m ranges from 1 to M, where M is the product of the stencil order and stencil dimension. The Manhattan distance-related features separate stencils with the same number of neighbors within each order. Compared to the assigned tensor, the feature set reflects the access pattern of a stencil in a more intuitive way.

TABLE 2: The candidate feature set of a stencil.

Feature	Meaning
$order$	The maximum extent of non-zeros.
nnz	The number of non-zeros in the tensor.
$sparsity$	The density of non-zeros in the tensor.
$nnz_{order-n}$	The number of non-zeros of order-n neighbors.
$nnzRatio_{order-n}$	The ratio of non-zeros of order-n neighbors.
nnz_{step-m}	The number of non-zeros of step-m locations.
$nnzRatio_{step-m}$	The ratio of non-zeros of step-m locations.

4.4 Performance Prediction

GSTuner uses pre-trained machine learning models to make performance predictions. We treat this prediction task as a regression problem: given a series of input features, predict the execution time of stencil computation. The input features include two parts: candidate feature set of a stencil and parameter setting in an OC. The parameter space includes parameters of numeric type (e.g., merging factor), Boolean type (e.g., shared memory usage) and enumeration type (e.g., streaming dimension) [21]. For the numerical parameters, we restrict their values to power of two inconsistent with existing works [8], [13]. We parameterize the range of the Boolean type as $\{0, 1\}$. We start from 1 with the unit stride to represent the parameters of the enumeration type. Note that when converted to input features, *GSTuner* performs $log2$ operation on the numerical parameters to ensure the stability of network training.

Unlike *StencilMART* that makes predictions for a specific OC, *GSTuner* targets at the global optimization space. Since the optimizations contained in different OCs are inconsistent, the input feature vector cannot be naively generated by concatenating parameters. As shown in Figure 7, *GSTuner* uniformly pre-processes the parameter settings among OCs into two data formats including zero-padding and varying-length. The zero-padding format reserves space for global optimizations in the feature vector, where the parameters not in OCs are padded with zero. The varying-length format groups parameters according to specific optimizations, and the parameter groups in each OC are chained into a sequence. The input with the above formats can be fed into the network structure, where zero-padding and varying-length

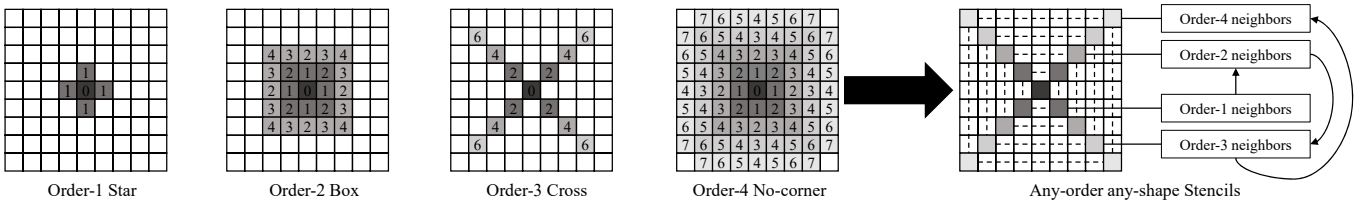


Fig. 6: An example of transforming the access pattern of a 2-D stencil into a sparse tensor, where the location of each non-zero element is assigned the Manhattan distance from the central element.

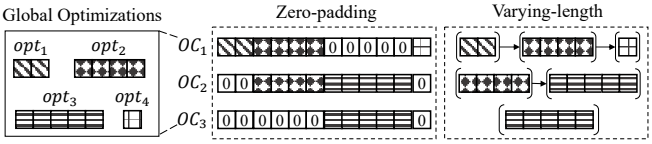


Fig. 7: Data pre-processing of feature vectors with inconsistent lengths across OCs.

are suitable for multilayer perceptron (MLP) and recurrent neural network (RNN), respectively.

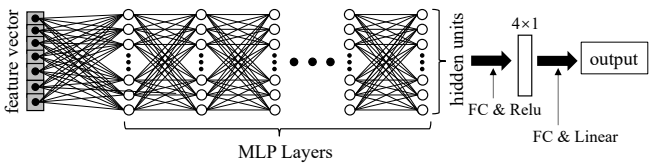


Fig. 8: The structure design of *ZPNet*.

GSTuner implements two regression mechanisms including *ZPNet* and *VLNet* for performance prediction. For *ZPNet*, the padded parameter setting is concatenated with the stencil feature set as the input feature vector. As shown in Figure 8, *ZPNet* comprises an input layer, multiple hidden MLP layers and an output layer that produces the predicted execution time for stencil computation. The number of MLP layers and the number of units per MLP layer can be adjusted to balance prediction performance and inference overhead. The input of *VLNet* includes varying-length parameter sequence and the stencil feature set. As shown in Figure 9, *VLNet* comprises a masking layer, a long short-term memory (LSTM) layer and multiple fully-connected (FC) layers. The masking layer exploits mask values to tell the LSTM layer to skip missing parameter groups when processing the data. The stencil feature set is fed into the FC layer, whose output and LSTM output are merged as joint features that flow into the subsequent FC layers.

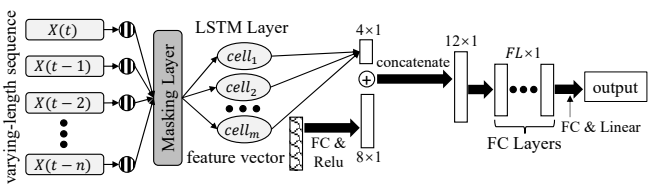


Fig. 9: The structure design of *VLNet*.

4.5 Search Space Sampling

For the trained regression model, the most straightforward idea is to traverse the optimization space and regard the parameter setting with the shortest prediction execution time as the optimum. However, the predicted optimum may fail to run at runtime due to resource spilling such as shared memory [21]. In addition, the predicted optimum usually differs from the actual optimum due to inevitable prediction errors and the similarity of high-performance settings. To address this issue, *GSTuner* utilizes the trained regression model to guide search space sampling, where the quota-based reward policy filters high-performance settings for subsequent evolutionary search. Search space sampling aims to avoid the actual execution of massive stencil instances, thus greatly reducing the auto-tuning cost.

Algorithm 2 illustrates the sampling process with the quota-based reward policy. The principle is that the sampling ratio of the OC with the optimum should be increased as a reward for its performance potential. At first, we initialize the sampling ratio of each OC (*SR*) according to the ratios of settings within OC to the total (Lines 3-4). The explicit constraints between optimization parameters [21] are checked when counting the number of settings in each OC. Next, we iteratively sample settings from OCs until a pre-set number (*N*) is reached (Line 5). Specifically, we reset the settings sampled in the last iteration (*iterSMP*), and then randomly sample without replacement from each OC based on *SR* (Lines 6-8). The sampling ratio of an empty OC is set to zero (Lines 9-10). We extract the OC with the predicted optimum in this iteration from *iterSMP* (Lines 13-14). We restrict the selection of the best OC to each iteration rather than all iterations evaluated currently, to avoid a single OC with an "early promise" dominating the entire sampling process.

We pre-define the total sampling ratio (*TR*) for one iteration, where *TR* equals to the summation of *SR*. We adjust the sampling ratio of each OC according to the prediction results. Specifically, the sampling ratios of other OCs are subtracted by the adjust ratio (*AR*), and then the remaining quota of *TR* is allocated to the best OC (Lines 15-24). Since the optimum is generally surrounded by worse settings [18], the optimal parameter setting may exist in OCs with poor mean performance. We set a lower limit (*LR*) for the sampling ratios of OCs, so that OCs that performed poorly in previous iterations always get a chance to "counterattack" with the optimum sampled. Finally, we append the settings sampled in this iteration to the total sample list (Line 25). The quota-based reward policy achieves the global exploration of the stencil optimization space and improves

Algorithm 2 Sampling with quota-based reward policy.

Input: number of settings to be sampled (N), number of OCs (nOC), settings of OCs ($ocSET$), sampling ratios of OCs (SR), total ratio (TR), adjust ratio (AR), lower ratio (LR)
Output: total settings sampled ($totSMP$)
// initialize SR as the ratios of OC settings to the total
$SR.initialize(ocSET.ratios(), TR)$
while $totSMP.count() < N$ **do**
  $iterSMP.reset([\,])$ // settings sampled in this iteration
  for id in range $[1, nOC]$ **do**
    $iterSMP.append([ocSET_{id}.sample(SR_{id})])$
    if $ocSET_{id}.empty()$ **then**
      $SR_{id} = 0.0$ // set the sampling ratio of empty OC to zero
    end if
  end for
  // index of the best performing OC predicted in this iteration
  $bpID = iterSMP.predict.argmin()$
  for id in range $[1, nOC]$ **do**
    if $id == bpID$ or $SR_{id} == 0.0$ **then**
      $continue$ // skip to the next OC
    end if
    if $SR_{id} - AR >= LR$ **then**
      $SR_{id} -= AR$ // reduce the sampling ratios of other OCs
    end if
  end for
  // allocate the remaining quota of TR to the best OC
  $SR_{bpID} = TR - SR.sum()$
  $totSMP.append(iterSMP.squeeze())$ // add sampled settings
end while

the sampling quality while ensuring the search breadth.

4.6 Evolutionary Search

Although the search space has been greatly narrowed after sampling, it is still time-consuming to use exhaustive search to determine the optimal parameter setting. Therefore, we propose an evolutionary search using genetic algorithm to find the optimal parameter setting efficiently. Figure 10 presents the multi-process genetic algorithm in *GSTuner*. As shown, multiple genes constitute an individual, which is evaluated by the fitness. Many individuals constitute a population, where the operations of each sub-population are handled by a process. The migration among the sub-populations is achieved using MPI communication. For migration, each sub-population exchanges individuals with its two neighborhoods (single-ring topology [29]).

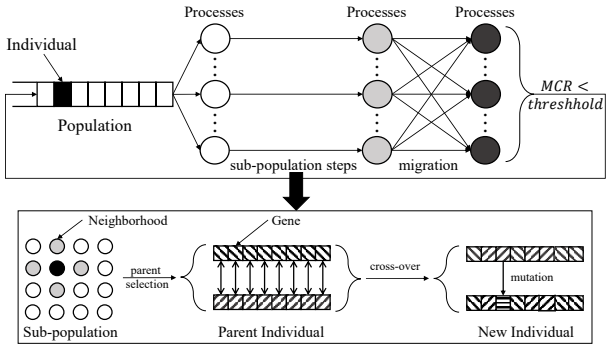


Fig. 10: Multi-process genetic algorithm in *GSTuner*.

The new individual in the sub-population is bred through uniform cross-over and mutation. The breeding involves three steps: *1)* the parents are selected from the four neighborhoods (higher fitness means higher selection chance); *2)* each gene of the individual is randomly chosen from the parents; *3)* the genes of the individual mutate with a certain probability (mutation rate). The mutation is used to prevent the individuals from falling into local optimum [30]. Genes are stored in binary, and the valid value range of each gene needs to be given in advance. However, the parameter values in the sampled search space are no longer continuous. To address this problem, we reindex the valid values of the parameter settings from the sampled search space. Assuming that the available value set of the parameter settings are $\{(0,1),(4,2),(3,4)\}$, we reindex the set to $\{0,2,1\}$ based on the ascending order. Then, the value range of the gene can be designated as $[0,2]$.

csTuner [21] takes into account the approximation of high-performance parameter settings, where the search process is stopped if the coefficient of variation (CV) of top-n fitness is less than a given threshhold. However, the appropriate threshhold varies with stencils, and a uniform threshhold may either increase the search cost or find sub-optimal parameter settings due to early stopping. Furthermore, calculating the CV for top-n fitness requires aggregating each subpopulation via MPI, which introduces considerable communication overhead. Instead, *GSTuner* adopts the subpopulation similarity as the termination condition. Specifically, the optimal parameter setting is determined when the ratio of the maximum-count individuals (*MCR*) in any subpopulation reaches a certain threshhold (*MCRT*). In such case, *GSTuner* achieves better stencil scalability and reduces the communication overhead within the population.

5 Evaluation

5.1 Experiment Setup

5.1.1 Hardware and Software Platforms

As shown in Table 3, we evaluate the effectiveness of *GSTuner* on a server that consists of Intel Xeon E5-2680 v4 CPU and two NVIDIA Tesla V100 GPUs. The experiments are conducted on Ubuntu 16.04 with GCC v7.5 and NVCC v10.1. The neural networks involved in *GSTuner* (i.e., *ZPNet* and *VLNet*) are built using TensorFlow release v1.15 [31].

TABLE 3: Hardware specifications.

	CPU	**GPU** $\times$ 2
Model	Intel Xeon E5-2680 v4	NVIDIA Tesla V100
Frequency	2.4GHz	1.5GHz
Cores	28	13440 (80 SMs)
Cache	32KB L1, 256KB L2, 35MB L3	6MB L2
Memory	378GB DDR4	32GB HBM2
Bandwidth	76.8GB/s	900GB/s

5.1.2 Stencil Programs and Datasets

We randomly generate 500 2-D and 500 3-D double-precision stencil programs using *GSTuner*, where the maximum stencil order is set to 4. The input grids of 2-D and 3-D stencils are set to $8,192^2$ and 512^3. We merge the OCs through PCCs and reduce the number of predicted OCs to 8. For each stencil program, we randomly select parameter settings from OCs and make measurements on GPU. After that, we obtain 211,766 2-D and 133,857 3-D stencil instances

to form the stencil dataset. The stencil dataset is randomly divided into a training set and a test set during cross validation (Section 5.1.4). To evaluate the effectiveness of auto-tuning, we explore 24 typical stencil programs (Section 3), which are a mixture of various patterns including stencil order, dimension, and shape. During sampling, the model trained on the stencil dataset is leveraged to predict the performance of typical stencil programs.

5.1.3 Search Methods and Implementation Details

We compare *GSTuner* with two popular stencil auto-tuning methods including *Artemis* [8] and *OpenTuner* [16]. For *GSTuner*, we set the sampling ratio of the global search space to 10%. In addition, *TR*, *AR* and *LR* (Algorithm 2) are set to 1%, 0.1% and 0.1%, respectively. For the genetic algorithm adopted in *GSTuner*, the number of sub-populations is set to 2, where each subpopulation contains 16 individuals. The cross-over rate and the mutation rate are set to 0.8 and 0.005, respectively. We extend *Artemis* and *OpenTuner* to support global optimization exploration by pre-processing parameter settings to the zero-padding format. For *OpenTuner*, we adopt the global genetic algorithm as the basis of its evolutionary technique. The options of the genetic algorithm are set to be consistent with *GSTuner*.

5.1.4 Comparison Metrics

We use the 5-fold cross validation method [28] to evaluate the accuracy of the models. For both *ZPNet* and *VLNet*, we select the Adam stochastic optimizer with a 0.0005 learning rate and a batch size of 256. We have fine-tuned the number of layers and the layer size. We take the mean absolute percentage error (MAPE) as the comparison metric for performance prediction. The key metric for determining the efficiency of auto-tuning methods is the amount of time required to obtain the optimal setting. Therefore, we compare *GSTuner* against *Artemis* and *OpenTuner* on iso-time search quality [32], where all methods are run until a fixed wall-clock time. To isolate the effects of randomness, we run each method 10 times and present the average results.

5.2 Results for Prediction

Figure 11 shows the testing error curves of *ZPNet* and *VLNet* during cross validation. As the number of training epochs increases, the test errors of both networks gradually converge to an almost constant value. The prediction results indicate that *GSTuner* efficiently extracts the features of stencil instances that contribute to execution time. We can also observe that *ZPNet* converges faster and achieves lower test errors than *VLNet*. This is mainly because stencil optimizations do not have the sequence relationship common in natural language processing, making it inappropriate to utilize LSTM to handle varying-length format. In contrast, *ZPNet* combines MLP and zero-padding format for better fitting, where the zero-padding format concatenates the global stencil optimizations to a fixed length.

Figure 12 shows the test errors of predicting typical stencils with the trained model. As seen, the test errors of stencils with diverse shapes, orders and dimensions are relatively stable. This indicates that the random stencil generator of *GSTuner* can generalize the computation patterns

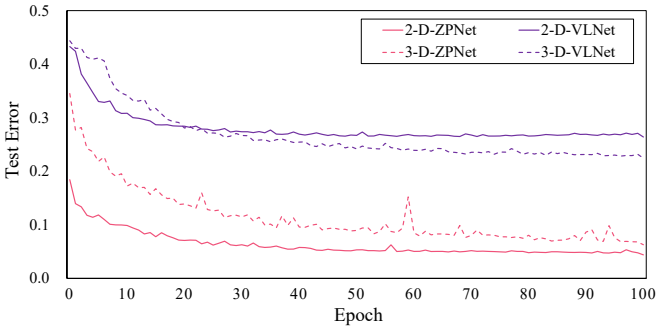


Fig. 11: Error curves of *ZPNet* and *VLNet* during testing.

with neighbor accesses by outputting a limited number of symmetric stencils. *ZPNet* still significantly outperforms *VLNet*, achieving average test errors of 6.1% and 28.6%, respectively. Furthermore, the inference time of *VLNet* is 7.6× that of *ZPNet* due to the computationally expensive LSTM processing. Considering the low test accuracy and high computation cost of *VLNet*, we will only adopt *ZPNet* in *GSTuner* for the following auto-tuning experiments.

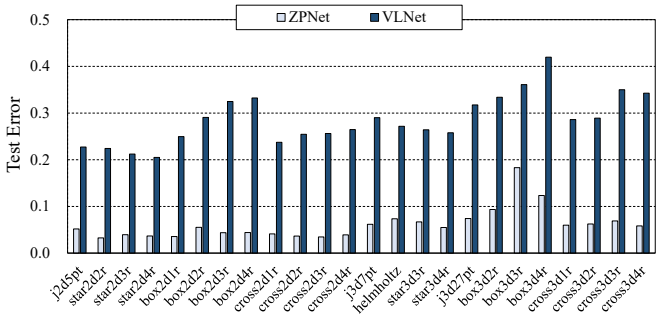


Fig. 12: Test errors of typical stencils with *ZPNet* and *VLNet*.

5.3 Results for Auto-tuning

Figure 13 shows the search range comparison between *GSTuner* and *Artemis* for typical stencils. As mentioned above, the sampling ratio of *GSTuner* is set to 10%, whereas *Artemis* divides the parameters into groups and performs hierarchical tuning on the parameter groups. As seen, the hierarchical mechanism of *Artemis* greatly reduces the search space, which is only 0.32× that of *GSTuner* on average. However, such aggressive pruning is hard to cover settings that meet performance requirements. *OpenTuner* searches the global parameter space, thus not shown in the figure.

Figure 14 shows the iso-time comparison between *GSTuner* and other auto-tuning methods for typical stencils, where the x-axis represents the elapsed time. The cutoff times for 2-D and 3-D stencils are set to 200 and 300 seconds, respectively. The missing points mean that the search process ends prematurely because it either exhausts potential parameter settings for search or reaches termination conditions (e.g., *MCR* in *GSTuner*). As seen, *GSTuner* has a better starting point and converges faster than *Artemis* and *OpenTuner*. This indicates that the reward policy of *GSTuner* guarantees the high quality of the sampled search space. *Artemis* spends less search time due to the fewer parameter

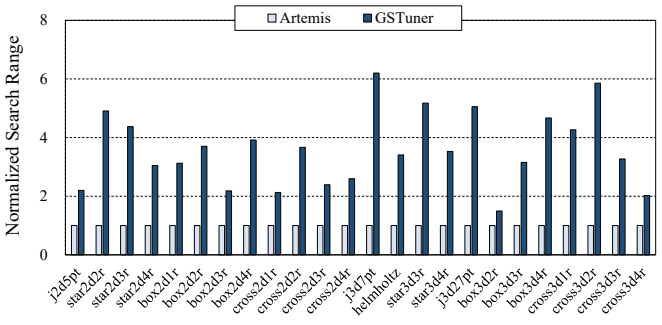


Fig. 13: Search range of *GSTuner* and *Artemis* normalized to *Artemis* on V100 GPU.

settings through hierarchical tuning. Since *OpenTuner* does not implement sampling methods, the large search space makes it difficult to converge in a short time.

Figure 15 shows the iso-time performance of auto-tuning methods normalized to *OpenTuner*. As seen, *GSTuner* outperforms *Artemis* and *OpenTuner* for most stencils. For the best performance found in iso-time evaluation, *GSTuner* achieves an average speedup of 1.5× and 1.4× over *Artemis* and *OpenTuner*, respectively. This proves that the feature extraction and evolutionary search adopted by *GSTuner* can be effectively generalized to diverse stencils. In contrast, *OpenTuner* tends to fall into a local optimum with small population size. *Artemis* achieves erratic performance due to experience-based hierarchical auto-tuning that may miss high-performance settings. For *box3d3r* stencil, *Artemis* even achieves 4.8× slowdown compared to *GSTuner*. In sum, *GSTuner* identifies better parameter settings with higher speed than other auto-tuning methods.

5.4 Applying to other GPU Hardware

To demonstrate the generality of our method, we evaluate *GSTuner* on another platform equipped with two NVIDIA Tesla A100 GPUs. Specifically, we recollect the stencil dataset on the new GPU hardware and reuse the *GSTuner* pipeline to quickly search for high-performance settings. Figure 16 shows the iso-time performance normalized to *OpenTuner* on A100 GPU. Again, *GSTuner* outperforms other auto-tuning methods for most stencils. *GSTuner* achieves 2.5× and 1.3× speedup on average over *Artemis* and *OpenTuner*, respectively. *OpenTuner* falls into a local optimum due to the large optimization space. *Artemis* maintains unstable performance due to its aggressive search space pruning. For example, *GSTuner* achieves comparable performance as *Artemis* for certain stencils (e.g., *star2d4r* and *star3d3r*), and significantly outperforms *Artemis* for others (e.g., 13.1× speedup for *box2d3r*). Note that the global regression mechanisms and reward-based sampling policy adopted in *GSTuner* do not require any expert knowledge. Therefore, *GSTuner* can be easily applied to various hardware platforms with stable auto-tuning quality.

5.5 Comparison with Previous Works

We extend *csTuner* and *StencilMART* to handle global optimization spaces represented in zero-padding format. After that, we compare *GSTuner* with *csTuner* and *StencilMART* in terms of iso-time performance and prediction accuracy, respectively. Figure 17 shows the iso-time performance normalized to *GSTuner* on V100 GPU. As seen, *GSTuner* outperforms *csTuner* for most stencils. Specifically, *GSTuner* achieves 1.1× speedup on average over *csTuner*. This is mainly because the reward mechanism of *GSTuner* effectively filters out poorly performing parameter settings. In addition, *GSTuner* takes subpopulation similarity as the termination condition, thereby avoiding the overhead of frequent MPI communication and determining parameter settings immediately when the algorithm converges.

Figure 18 shows the test errors of predicting typical stencils with *StencilMART* and *GSTuner*. *StencilMART* and *GSTuner* adopt the same MLP-based network structure, where the difference lies in the stencil feature sets fed into the model. *GSTuner* extends the original candidate set with Manhattan distance-related features. As seen, *GSTuner* outperforms *StencilMART* for all stencils, achieving average test errors of 6.1% and 9.7%, respectively. The reason is that the Manhattan distance-related features separate stencils with the same number of neighbors within each order, and thus more precisely represent the neighboring access patterns for stencil computation.

5.6 Adapting to other Grid Sizes

We have demonstrated in Section 3.3 that the parameter settings adapted to the target stencil and underlying hardware are relatively stable within a certain range of grid sizes. Since only the relative performance is required for search space sampling, we discuss whether the trained model of one grid size can be directly adapted to other sizes. Figure 19 shows the iso-time performance of auto-tuning methods, where the models used by *GSTuner* are trained with stencil dataset of $8,192^2$ (2-D) and 512^3 (3-D) sizes. It can be observed that *GSTuner* outperforms other auto-tuning methods for most stencils. For $2,048^2$ (2-D) and 128^3 (3-D) sizes, *GSTuner* achieves an average speedup of 1.5× and 1.5× over *Artemis* and *OpenTuner*, respectively. For $4,196^2$ (2-D) and 256^3 (3-D) sizes, *GSTuner* achieves 2.5× and 1.4× speedup on average. The results indicate that when varying grid sizes, *GSTuner* can reuse the model to achieve superior auto-tuning performance without recollecting the stencil dataset.

5.7 Parameter Sensitivity Analysis

5.7.1 Network Depth and Width

Figure 20 shows the test errors of *ZPNet* as we vary the number of MLP layers along with their size. The x-axis represents the layer size ranging from 2^4 to 2^{10} with a stride of ×2. As seen, *ZPNet* for 2-D and 3-D stencils appears to follow a similar error trend. Specifically, increasing the number of layers and their sizes leads to lower test errors. In addition, increasing the number of layers beyond seven leads to diminishing returns for improving prediction accuracy. Therefore, we can conclude that using seven MLP layers for *ZPNet* is a reasonable setting. *GSTuner* provides an easy-to-use interface for modifying network parameters to evaluate the impact of network designs.

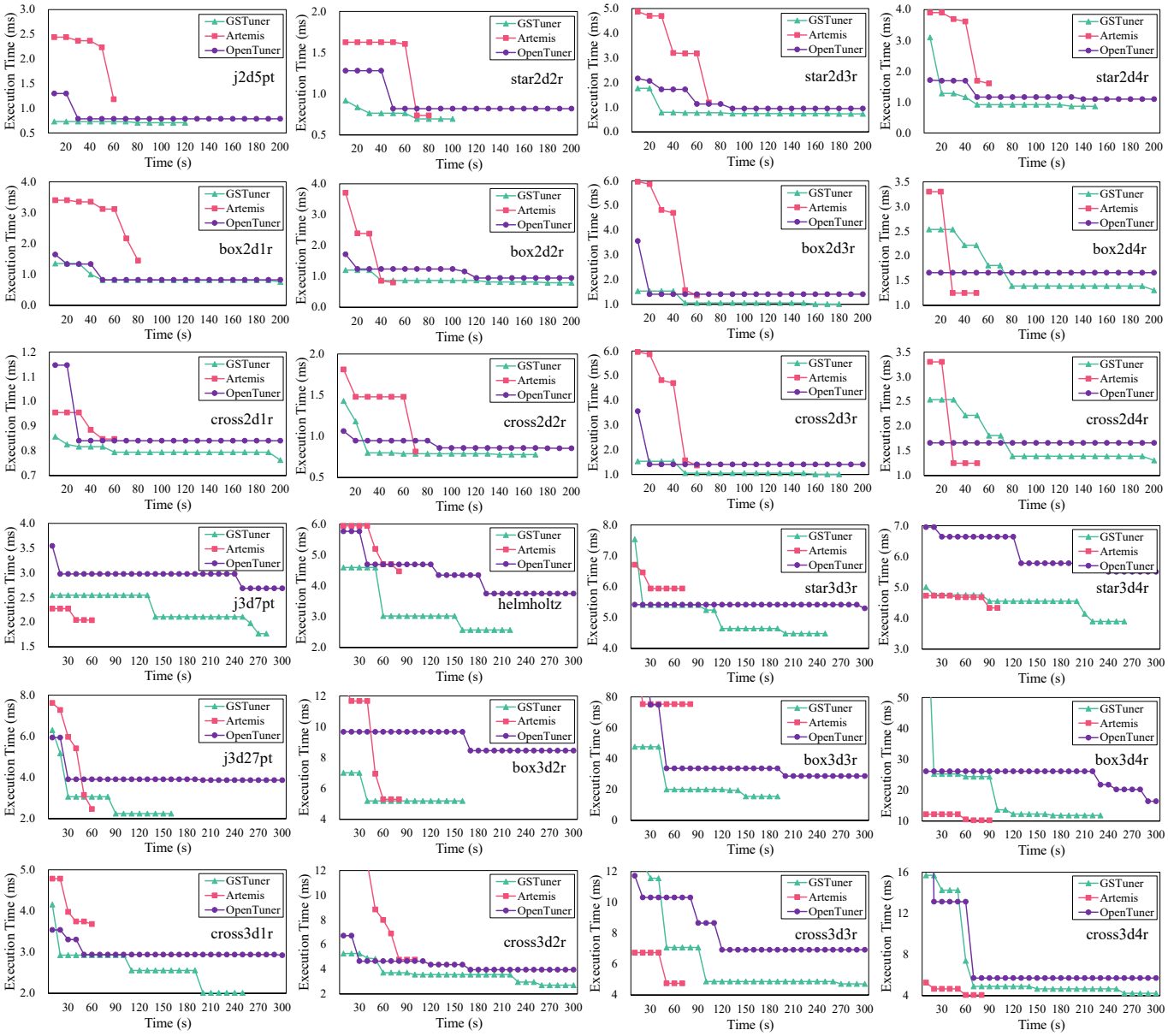


Fig. 14: Iso-time comparison of stencil auto-tuning methods on V100 GPU.

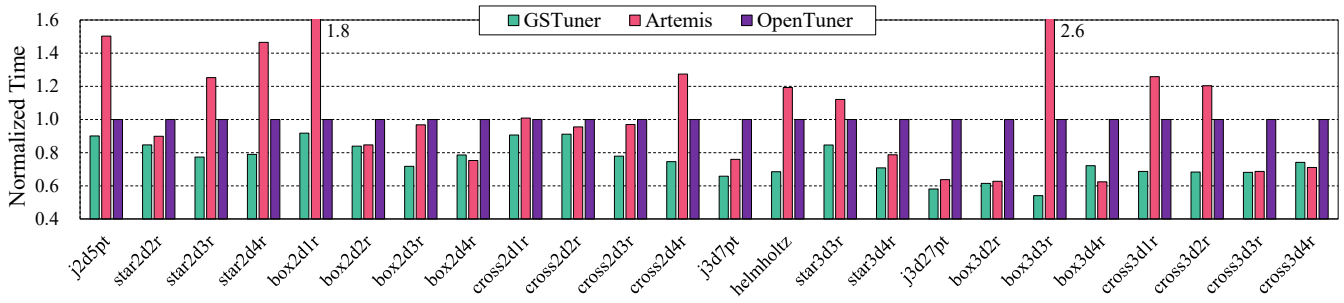


Fig. 15: Iso-time performance of auto-tuning methods normalized to *OpenTuner* on V100 GPU.

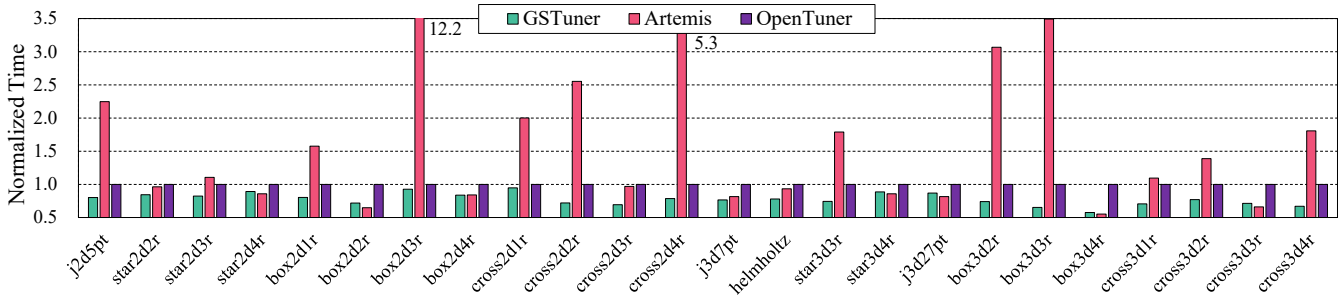


Fig. 16: Iso-time performance of auto-tuning methods normalized to *OpenTuner* on A100 GPU.

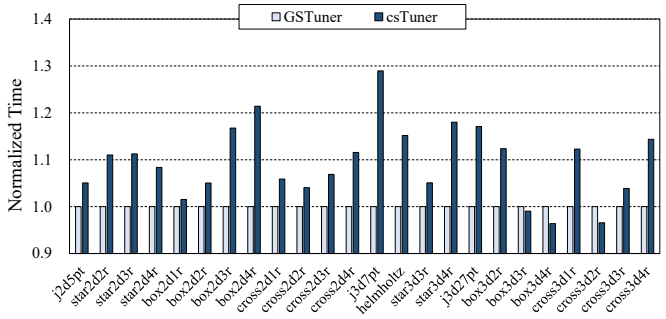


Fig. 17: Iso-time performance of *csTuner* and *GSTuner* normalized to *GSTuner* on V100 GPU.

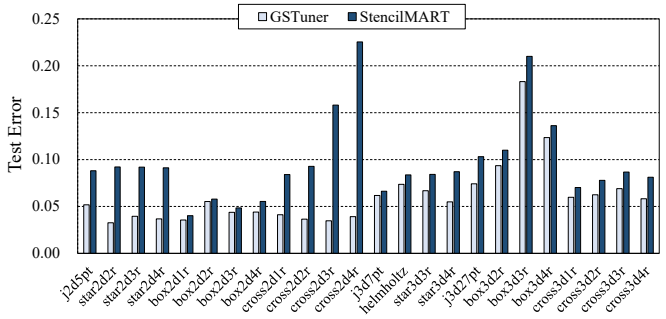


Fig. 18: Test errors of typical stencils with *StencilMART* and *GSTuner* on V100 GPU.

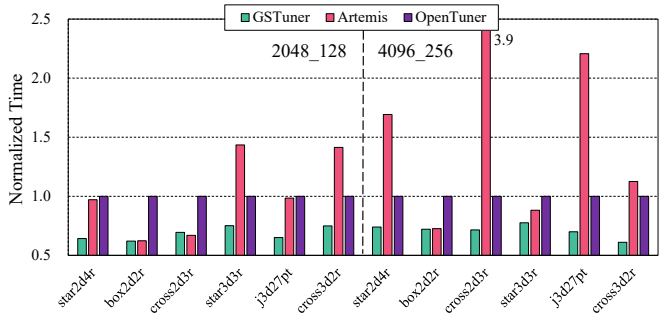


Fig. 19: Iso-time performance of auto-tuning methods for other grid sizes. The models used by *GSTuner* are trained with stencil dataset of $8,192^2$ (2-D) and 512^3 (3-D) sizes.

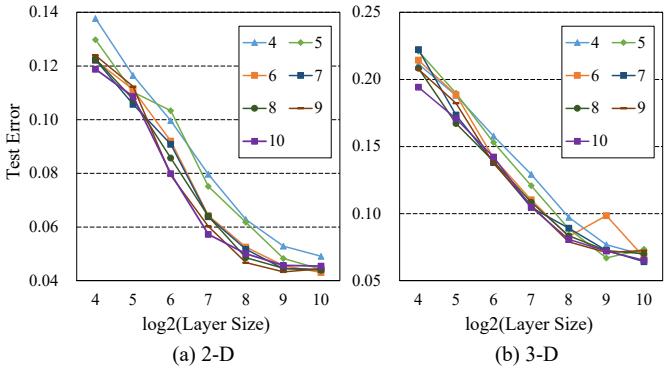


Fig. 20: Test error of *ZPNet* as we vary the number of hidden layers and layer size. The x-axis is in a logarithmic scale.

5.7.2 Sampling Ratio

Figure 21 shows the iso-time performance of *GSTuner* with different sampling ratios, where *SR-X* means that the sampling ratio is X. In theory, a smaller sampling ratio completes the search process faster, but the limited search range may miss the optimal setting. In contrast, a larger sampling ratio is more promising to find the optimal setting, yet with a longer search time. As seen, *GSTuner* with *SR*-20% outperforms that with *SR*-10% for 14 out of 24 stencils. However, *GSTuner* with *SR*-20% achieves unstable performance, achieving more than 1.1× slowdown compared to that with *SR*-10% for certain stencils (e.g., *j3d7pt* and *box3d3r*). This further indicates that *GSTuner* effectively filters out poor-performing settings through the reward-based policy at low sampling ratios. Therefore, we select *SR*-10% for *GSTuner* to balance the search space and search speed.

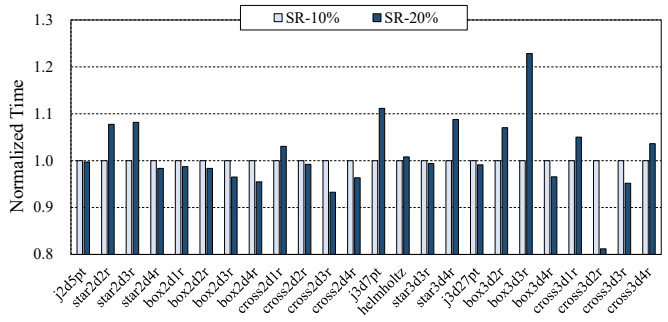


Fig. 21: Iso-time performance of *GSTuner* with different sampling ratios. *SR-X* means that the sampling ratio is X.

5.7.3 MCR Threshhold

Figure 22 shows the final tuning performance and search cost of *GSTuner* with different *MCRT*s normalized to that with 0.75 *MCRT*. *MCRT_X_ET* and *MCRT_X_ST* indicate the normalized execution time and search time when *MCRT* is set to X. A good *MCRT* enables *GSTuner* to quickly terminate the search process after finding the optimal setting. As seen, *GSTuner* with 0.5 *MCRT* achieves the same performance as that with 0.75 *MCRT* for stencils except for *j3d27pt*. In addition, *GSTuner* with 0.25 *MCRT* is inferior to that with 0.75 *MCRT* for 8 stencils, and even achieves over 1.1× slowdown for *cross3d2r* and *cross3d3r*. Whereas for search cost, *GSTuner* with either 0.5 or 0.25 *MCRT*s completes faster for most stencils, achieving up to 2.2× and 1.3× lower time. The results demonstrate the stencil scalability of *MCR* termination conditions. We select 0.5 *MCRT* for *GSTuner* considering performance stability.

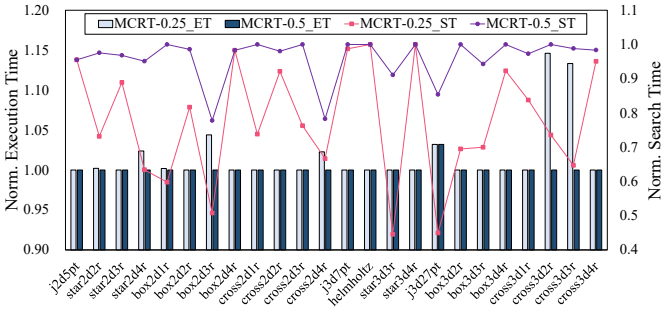


Fig. 22: Final performance and search cost of *GSTuner* with different *MCRT*s normalized to that with 0.75 *MCRT*. *MCRT_X_ET* and *MCRT_X_ST* indicate the normalized execution time and search time when *MCRT* is set to X.

5.8 Overhead Analysis

We discuss the overhead of *GSTuner* from both offline and online aspects. It takes about 8.5 and 9.1 hours respectively to collect 2-D and 3-D stencil datasets on our experiment server. In addition, it takes 8.1 and 5.2 minutes respectively to train *ZPNet* with 2-D and 3-D stencil datasets. Since dataset collection and network training only need to be done offline once, we do not consider them in the overhead analysis of online auto-tuning. The online cost of *GSTuner* can be divided into two parts pre-processing and search process. The pre-processing can be further divided into stencil feature extraction and search space sampling. Search space sampling contains reward-based setting selection and performance prediction with network inference. Note that the pre-processing overhead of *GSTuner* has been taken into account in the iso-time experiments.

Since the search process dominates the online cost of *GSTuner*, Figure 23 shows the pre-processing time normalized to the search process. *GSTuner* adopts the genetic algorithm with *MCR* termination condition in the search process. As seen, the pre-processing time is negligible compared to the search process, occupying only 0.8% of the search time on average. Moreover, the pre-processing overhead of *GSTuner* can be further amortized for the larger search space with more stencil optimizations proposed in the future.

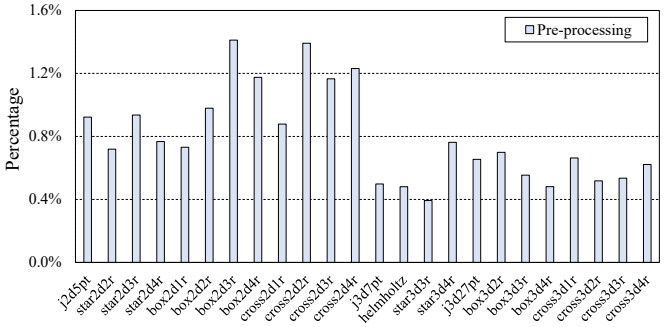


Fig. 23: Pre-processing time of *GSTuner* normalized to the search process.

6 Related Work

Stencil DSLs and Optimizations. Based on the regular patterns of stencil computation, existing research works exploit the integration of optimization schemes into DSLs to achieve automatic code transformation and optimization [5], [7], [9], [13], [14], [15], [33], [34], [35]. *Physis* [9] translated user-written stencil code into scalable implementation for GPU-equipped cluster. *Forma* [5] proposed a DSL for image processing application with stencil operations. Hagedorn *et al.* [7] explored how to use *LIFT* primitives to implement stencil codes and optimizations such as tiling. *AN5D* [13] implemented high-degree temporal blocking and spatial blocking, in addition to low-level optimizations to reduce the resource usage. Li *et al.* [33] proposed spatial computation folding to reduce data conflicts and optimized the vectorization with shifts reusing. *TCStencil* [15] exploited tensor cores to accelerate stencil computation by refactoring it into reduction and summation operations. Ahmad *et al.* [35] presented stencil algorithms for linear stencils based on random walks and Gaussian approximation. The above works lack effective support for parameter auto-tuning. To address the limitations of above works, *GSTuner* can be integrated into these DSLs to determine the optimal settings for target optimizations quickly.

Performance Auto-tuning on GPUs. Since identifying the optimal kernel variants is extremely challenging for both programmers and code generators, a large amount of research works focus on the auto-tuning of target problems on GPUs [22], [36], [37], [38], [39], [40], [41], [42], [43]. Kurzak *et al.* [36] proposed heuristic auto-tuning to prune the search space and generate the fastest code variant of matrix multiplication kernels. Li *et al.* [37] resolved the conflict between concurrency and register usage by precomputing the critical points and selecting the global optimum. Pfaffe *et al.* [40] integrated hierarchical online auto-tuning with polyhedral parallelization to reduce search complexity and increase convergence speed. *Ansor* [22] sampled optimization combinations and utilized an evolutionary search with a learned cost model to fine-tune the tensor programs. *LLAMA* [41] traversed large spaces by dynamically running a cost-based optimizer and configuring individual operation invocations. Sun *et al.* [43] jointly learned the structural and statistical features via the graph attention network to find the optimal code implementations. The above works are orthogonal to this paper that targets the auto-tuning of stencil kernels. In

turn, *GSTuner* can be extended to other target programs due to its global search capability.

7 Conclusion and Future Work

In this paper, we propose an adaptive auto-tuning framework *GSTuner*, which efficiently identifies the optimal parameter setting of the global optimization space for stencils on GPUs. *GSTuner* represents stencil patterns as neighboring features and unifies feature vectors of OCs for model training. After that, *GSTuner* leverages the quota-based reward policy and trained models to guide the sampling process of the global space. Finally, *GSTuner* adopts the genetic algorithm regarding sub-population similarity to conduct the parameter search of the sampled space. The experiment results show that *GSTuner* can accurately predict the execution time of stencil instances on GPUs. In addition, *GSTuner* can identify high-quality parameter settings in a shorter time compared to the state-of-the-art works.

A potential limitation of *GSTuner* is that it cannot support complex stencils with boundary conditions or kernel dependencies. Complex stencils are hard to be represented by a single sparse tensor, which is beyond the scope of the random stencil generator. In such case, we need to quantify the impact of boundary conditions and kernel dependencies on performance, and then extract representative features to complement the stencil candidate feature set. Furthermore, the versatility of *GSTuner* allows it to be extended for multi-GPU scenarios. After collecting and parameterizing the optimization collection on multiple GPUs, we can reuse the *GSTuner* pipeline for performance auto-tuning.

Acknowledgments

This work is supported by National Key Research and Development Program of China (Grant No. 2022ZD0117805), National Natural Science Foundation of China (Grant No. 62072018, 62322201 and U22A2028), the Fundamental Research Funds for the Central Universities (Grant No. 2462023YJRC023 and YWF-23-L-1121), and Iluvatar CoreX semiconductor Co., Ltd. Hailong Yang is the corresponding author.

References

[1] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, 2013.

[2] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar, "Local recovery and failure masking for stencil-based applications at extreme scales," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[3] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 429–443, 2015.

[4] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–12.

[5] M. Ravishankar, J. Holewinski, and V. Grover, "Forma: A dsl for image processing applications to target gpus and multi-core cpus," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, 2015, pp. 109–120.

[6] C. Oh, Z. Zheng, X. Shen, J. Zhai, and Y. Yi, "Gopipe: a granularity-oblivious programming framework for pipelined stencil executions on gpu," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 43–54.

[7] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 100–112.

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

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

[10] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 66–75.

[11] Q. Sun, Y. Liu, H. Yang, Z. Jiang, Z. Luan, and D. Qian, "Stencilmart: Predicting optimization selection for stencil computations across gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 875–885.

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

[13] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "An5d: automated stencil framework for high-degree temporal blocking on gpus," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 199–211.

[14] M. Li, Y. Liu, Y. Hu, Q. Sun, B. Chen, X. You, X. Liu, Z. Luan, and D. Qian, "Automatic code generation and optimization of large-scale stencil computation on many-core processors," in *Proceedings of the 50th International Conference on Parallel Processing*, 2021, pp. 1–12.

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

[16] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[17] J. D. Garvey and T. S. Abdelrahman, "Automatic performance tuning of stencil computations on gpus," in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 300–309.

[18] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1280–1295.

[19] V. Martínez, F. Dupros, M. Castro, and P. Navaux, "Performance improvement of stencil computations for multi-core architectures based on machine learning," *Procedia Computer Science*, vol. 108, pp. 305–314, 2017.

[20] B. Cosenza, J. J. Durillo, S. Ermon, and B. Juurlink, "Autotuning stencil computations with structural ordinal regression learning," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 287–296.

[21] Q. Sun, Y. Liu, H. Yang, Z. Jiang, X. Liu, M. Dun, Z. Luan, and D. Qian, "cstuner: Scalable auto-tuning framework for complex stencil computation on gpus," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 1–12.

[22] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 863–879.

[23] Q. Sun, L. Yi, H. Yang, M. Li, Z. Luan, and D. Qian, "Qos-aware dynamic resource allocation with improved utilization and energy efficiency on gpu," *Parallel Computing*, vol. 113, p. 102958, 2022.

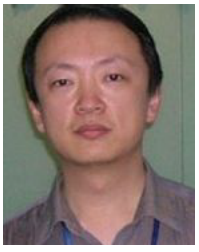
[24] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on gpus," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 168–182.
[25] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 65–76.
[26] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.
[27] A. Zheng and A. Casari, *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.", 2018.
[28] Q. Sun, Y. Liu, M. Dun, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "Sptfs: sparse tensor format selection for mttkrp via deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.
[29] Z. Xiao, X. Liu, J. Xu, Q. Sun, and L. Gan, "Highly scalable parallel genetic algorithm on sunway many-core processors," *Future Generation Computer Systems*, vol. 114, pp. 679–691, 2021.
[30] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.
[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.
[32] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 943–958.
[33] K. Li, L. Yuan, Y. Zhang, and Y. Yue, "Reducing redundancy in data organization and arithmetic calculation for stencil computations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
[34] X. You, H. Yang, Z. Jiang, Z. Luan, and D. Qian, "Drstencil: Exploiting data reuse within low-order stencil on gpu," in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. IEEE, 2021, pp. 63–70.
[35] Z. Ahmad, R. Chowdhury, R. Das, P. Ganapathi, A. Gregory, and Y. Zhu, "Brief announcement: Faster stencil computations using gaussian approximations," in *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, 2022, pp. 291–293.
[36] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemm kernels for the fermi gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, 2012.
[37] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. Chavarría-Miranda, and H. Corporaal, "Critical points based register-concurrency autotuning for gpus," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1273–1278.
[38] R. Lim, B. Norris, and A. Malony, "Autotuning gpu kernels via static and predictive analysis," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 523–532.
[39] J. Dongarra, M. Gates, J. Kurzak, P. Luszczek, and Y. M. Tsai, "Autotuning numerical dense linear algebra for batched computation with gpu hardware accelerators," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2040–2055, 2018.
[40] P. Pfaffe, T. Grosser, and M. Tillmann, "Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 354–366.
[41] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 1–17.
[42] X. Zhang, J. Xiao, and G. Tan, "I/o lower bounds for auto-tuning of convolutions in cnns," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 247–261.
[43] Q. Sun, X. Zhang, H. Geng, Y. Zhao, Y. Bai, H. Zheng, and B. Yu, "Gtuner: tuning dnn computations on gpu via graph attention network," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1045–1050.

Qingxiao Sun is a PhD student in School of Computer Science and Engineering, Beihang University. He is currently working on GPU hardware extension and performance optimization. His research interests include computer architecture, HPC and deep learning.

Yi Liu is a professor in School of Computer Science and Engineering, and Director of the Sino-German Joint Software Institute (JSI) at Beihang University, China. In 2000, he completed Ph.D in Department of Computer Science of Xi'an Jiaotong University. His research interests include computer architecture, HPC and new generation of network technology.

Hailong Yang is an associate professor in School of Computer Science and Engineering, Beihang University. He received the Ph.D degree in the School of Computer Science and Engineering, Beihang University in 2014. He has been involved in several scientific projects such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency.

Zhonghui Jiang received the B.S. degree in School of Mechanical and Aerospace Engineering, Jilin University. He is currently pursuing the M.S. degree in School of Computer Science and Engineering, Beihang University. His research interests include GPU performance optimization, deep learning compiler and machine learning system.



Zhongzhi Luan received the Ph.D. in the School of Computer Science of Xi'an Jiaotong University. He is an Associate Professor of Computer Science and Engineering, and Assistant Director of the Sino-German Joint Software Institute (JSI) Laboratory at Beihang University, China. Since 2003, His research interests including distributed computing, parallel computing, grid computing, HPC and the new generation of network technology.

Depei Qian is a professor at the Department of Computer Science and Engineering, Beihang University, China. He received his master degree from University of North Texas in 1984. He is currently serving as the chief scientist of China National High Technology Program (863 Program) on high productivity computer and service environment. He is also a fellow of China Computer Federation (CCF). His research interests include innovative technologies in distributed computing, high performance computing and computer architecture.